Bolt Beranek and Newman Inc.

Report No. 3607

Intelligent On-Line Assistant and Tutor System

18 September 1975 to 31 January 1977

Prepared for:
Advanced Research Projects Agency

(14) BBN [redacted] No. 3607

(11)

(6) INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM.

(9) FINAL TECHNICAL REPORT.

18 SEPT [redacted] 75 - 31 JAN [redacted] 77,

(10) WILLIAM L. ASH,
ROBERT J. BOBROW,
MARIO C. GRIGNETTI
ALICE K. HARTLEY

(11) 31 Jan 77

(12) 119 p.

PREPARED FOR:

(15) ADVANCED RESEARCH PROJECTS AGENCY
N00014-76-C-0476, ARPA ORDER [redacted]-3091

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. N00014-76-C-0476.

060 100

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: BBN Report No. 3607
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM

5. TYPE OF REPORT & PERIOD COVERED: FINAL REPORT 18-SEP-75 to 31-JAN-77

6. PERFORMING ORG. REPORT NUMBER: BBN Report No. 3607

7. AUTHOR(s): ASH, William L.; BOBROW, Robert J.; GRIGNETTI, Mario C.; HARTLEY, Alice K.

8. CONTRACT OR GRANT NUMBER(s): N00014-76-C-0476

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt, Beranek and Newman, Inc. 50 Moulton Street Cambridge, MA 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 6D30

11. CONTROLLING OFFICE NAME AND ADDRESS: ONR Department of the Navy Arlington, VA 22217

12. REPORT DATE:

13. NUMBER OF PAGES:

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Dept. of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence, Computer Assisted Instruction, Natural Language Interfaces, Syntactic Analysis, Semantic Interpretation, LISP, INTERLISP development

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

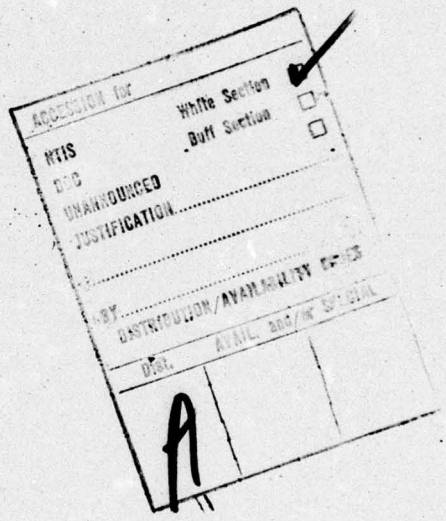
This report describes work performed to develop a system to help people who are not computer experts deal with their Intelligent Terminal, an envisioned personal computer of great sophistication and power. The system was aimed at being capable of mediating "intelligently" between these users and the tools that could be available to them in the Intelligent Terminal: provide streamlined instruction on how to use these tools, answer questions posed in English about the tools, and request that certain actions described in (OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

English be performed using the tools. The system is written in INTERLISP.

The report describes in detail the work performed to bring up INTERLISP on a DEC-PDP-11 computer, and the work performed on an Intelligent oN-Line Assistant and Tutor (INLAT) for new users of HERMES, an independently developed message processing system that exemplifies the kind of tool that could be made available in future Intelligent Terminals.



INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM

Final Technical Report

18 September 1975 to 31 January 1977

ARPA Order No. 3091

Program Code No. 6D30

Name of Contractor:
Bolt Beranek and Newman Inc.

Effective Date of Contract:
18 September 1975

Contract Expiration Date:
31 December 1976

Amount of Contract:
$592,562.00

Contract No. N00014-76-C-0476

Principal Investigator:
Mario C. Grignetti
(617)491-1850 x394

Scientific Officer:
Gordon Goldstein

Short Title of Work:
INTELLIGENT ON-LINE ASSISTANT AND TUTOR (INLAT)

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 3091

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

We are interested in making it easy for computer-naive people to sit in front of a computer terminal (eventually an Intelligent Terminal) and learn how to use the computer tools available through it. We believe that a good way to achieve this goal is by means of a reactive learning environment, where tutorial services ranging from systematic teaching to occasional on-line help are made easily and immediately available to users.

The three fundamental aspects that such a learning environment must incorporate are:

1) well-organized tutorials, i.e. a set of lessons that systematically teach the how-to knowledge that is essential for useful operation of a computer tool;
2) the ability to "look" over the user's shoulder and be "aware" of what he is doing, so that when needed, the system can offer help that is specific to the task the user is engaged in.
3) the ability to perform these services in response to user requests expressed in natural language (English).

Since such reactive learning environments incorporate Artificial Intelligence techniques, and to the extent that they appear to have features that mimic human intelligence, we have coined for them the acronym INLAT ("Intelligent" oN-Line Assistant and Tutor).

The specific objectives of the present contract were to develop INLAT's for three of the tools that may conceivably be the mainstays of future Intelligent Terminals. These tools are Hermes

(a computer mail processing system), the Rand editor (a multi-window, scope editor) and a News Service information retrieval system (such as the New York Times Information Bank or Stanford's News Service program).

INLAT's can be best developed in a programming environment that makes it easy to turn out large and sophisticated systems in relatively short times. It is very important, especially when dealing with potentially large populations of computer-naive users, to be able to drastically revise design concepts, approaches, and implementations. We believe that INTERLISP is the best available "milieu" in which this fast turn-over development effort can take place. Not only is it based on a language of enormous power and flexibility (LISP), the work horse of most AI work today; it is also a system that has "built in" many of the facilities (such as the Do What I Mean (DWIM) and the Programmer's Assistant) that are considered essential in an Intelligent Terminal. For these reasons, the crucial part of the present effort is to provide an INTERLISP environment on the machine that has been selected as a first prototype for the Intelligent Terminal, -- the DEC PDP-11.

The body of the Report begins with a description of our design goals and the work performed in bringing up an INTERLISP system on the PDP-11. In the remainder of the Report we describe the work performed to implement our INLAT. Although the INLAT is intended as an aid in the use of multiple tools, the bulk of our work in this area to date has been centered on providing a counselor and assistant for the Hermes message system being

developed independently at BBN. Therefore, our description is limited to an INLAT-Hermes system.

SECTION 2 - INTERLISP-11

Introduction and Purpose

Although INTERLISP is a formidably powerful tool, and its advantages for AI work are widely recognized, its usage has been relegated to, and is commonly associated with powerful, large machines. In recent years, however, the technological advances in computer hardware have been so dramatic that today's "minicomputers" have almost equalled the power of such mainstays of LISP as the PDP-10, and are available at a fraction of the latter's cost. Therefore, our goal is to bring up a full INTERLISP system (as it now exists in the PDP-10 under TENEX) on a PDP-11 computer. This we call INTERLISP-11.

More specifically, what we want to provide is an INTERLISP environment with the following characteristics:

a) serves a single user,

b) is fully compatible with INTERLISP,

c) runs at half the speed of INTERLISP-10 when there is only one user (therefore it should appear much faster than INTERLISP-10 under "normal" TENEX load conditions),

d) has a larger address space (4 million words) than INTERLISP-10 (The INTERLISP-10 address space of 256K words is becoming confining for present day applications), and

e) minimizes machine dependent code for easier exporting to another computer.

In the rest of this section we describe our plan for achieving these purposes. We begin by describing the hardware configuration that we selected and the reasons for choosing it. Next we describe the software work that is needed to accomplish our goals with the selected hardware.

Hardware

The characteristics of the desired INTERLISP-11 environment impose three kinds of broad requirements on the hardware. These requirements comprise the ability for:

1) efficiently executing LISP compiled code
2) storing LISP data structures compactly. In INTERLISP-10 we have 36 bit words, 18 bit pointers, and lots of instructions for manipulating half words. On the PDP-11 we have 16 bit words and 22 bit pointers. In the simplest case, one would like to pack 2 pointers into 3 words and in some cases (e.g. lists) even more compact encodings are possible.
3) dealing with a large virtual address space,

These requirements impose a particular selection within the PDP-11 family and the incorporation of extra hardware. Specifically, the hardware configuration that was specified for our purpose consists of a DEC PDP-11/40 processor with 128K words of core memory, a fast 512K words fixed head disk for swapping (RS04), and a slower 28M words disk for secondary swapping and filing (Telefile). In addition, we have an interface for network communications (IMP-11A), a Writable Control Store (WCS) for

efficient and compact microcoding of compiled LISP code and for compact list structure, a specially designed Memory Mapping Device (MMD) to facilitate the implementation of the large address space, and a number of other peripheral devices. In what follows we will describe only the two non-standard components in the above configuration, the WCS and the MMD.

Writable Control Store -- Purpose and Description

The Writable Control Store (WCS) was designed at Carnegie Mellon University by Professor Samuel Fuller and his group, and it is used extensively in their Hydra project. It provides a way of augmenting the capabilities of the standard 11/40 processor which is crucially important for our purposes. The most important advantage derived from using the WCS is that it allows us to design an instruction set for LISP-compiled code that is independent of the host machine instruction set and is therefore better suited to the requirements of LISP. The resultant object programs are therefore more compact and run more efficiently than their PDP-10 counterparts. In terms of required memory size, we estimate that object programs written in the LISP instruction set are only 1/3 as large (in terms of number of bits) as equivalent PDP-10 compiled LISP. Alternately, they require 3/4 as many 16-bit PDP-11 words as required on the PDP-10 (36-bit words). Finally, because of its relative machine independence, the new object code permits substantially easier transfer to newer, more cost-effective hardware when it becomes available.

The 11/40 processor is implemented with 256 words (56 bits) of read-only microcode. It is designed so that the 11/40 options, such as the Extended Instruction Set and the Floating Point instructions, can be installed by simply plugging in additional microcode read-only memory modules. This plug-in extensibility makes it possible to design and build the WCS.(1)

The WCS has the following characteristics:

1) it provides 1024 words of additional high speed microcode control store which can be used to implement critical parts of INTERLISP-11, including the object code execution "interpreter";
2) it is writable, facilitating both design and debugging, and allowing for the possibility of swapping microcode sets (such as one for the garbage collector);
3) it provides an extension of the underlying PDP-11 micro machine, enhancing its generality. The standard PDP-11 ROM has 56-bit words while the extended micro machine is controlled by 80-bit words.

While (1) and (2) are fairly clear, (3) requires somewhat more comment. Simply adding extra control store to the basic 11/40 micro machine would not provide a very useful general-purpose microprogrammed machine, since the 11/40 microcode is designed primarily for implementation of the PDP-11 instruction set and thus lacks generality. The WCS adds to it the following features

(1) The 11/40 is the only PDP-11 able to accommodate the WCS

a) Expands the micro program address space to 2048 words. Words 0-255 are the PDP 11/40 ROM. Words 256-287 are the PROM, a programmable ROM that is required for reading and writing the RAM. Words 288-1023 do not exist. Words 1024 to 2047 are the RAM, the program storage for the WCS.

b) A 16 bit mask/shift capability that allows extraction of any contiguous field at any position in the result. The result may be used either for arithmetic operations or for multiway branch control. The shifter performs a 0-15 position right rotate, and the mask can clear 0-15 bits from either or both ends of the word. This field extraction capability is particularly useful for instruction decoding as it allows an N-way dispatch on any field to be performed in one micro-instruction.

c) A microsubroutine capability

d) The ability to specify a 16 bit constant for masking, arithmetic, and generating micro-subroutine return addresses (The only generally useful constants provided by the 11/40 are 1, 2 and 4).

e) A 16 word (x 16 bit) stack for temporary values and for microsubroutine return addresses.

f) Arithmetic carry control for multiple precision arithmetic.

The extra power provided by these facilities allowed us to design an instruction set that is almost ideally suited for LISP compiled code.

Memory Mapping

The 4M words address space specified previously for INTERLISP-11 clearly exceeds the amount of main memory (core or RAM) one can reasonably expect will be available for Intelligent Terminal applications. This implies that a way must be found to map the INTERLISP-11 address space into a smaller real memory.

The existing mapping device for the PDP-11 (the so called Memory Management Option) doesn't help us very much since it is designed for the opposite application it maps a small virtual address space into a larger amount of real memory. This is good for multiuser applications since it allows several user processes with small virtual spaces (32k) to be resident in core simultaneously, but this is not our case.

We considered the software approach: a page table that lives in core, microprogramming for putting together a real core address, and reloading of Memory Management Registers (MMR's) for accessing that real address. This approach is undesirable for several reasons: a) it is slow, b) the microcoded instructions that we'd need would usurp WCS memory space that could be used for other purposes, c) and it would impose an extra burden on our programming tasks. This last consideration is very important; in order to minimize reloading of MMR's, which is very expensive, we would have to be very clever in order to keep core "windowed in" as much as possible. Clearly, a solution that allows us to treat all core references equally is much preferred, since it simplifies the programming effort, reduces debugging, and speeds coding.

These reasons suggested a hardware/software trade-off: a simple hardware pager, that we call Memory Mapping Device (MMD), to map the 22-bit address space of INTERLISP-11 into a 19 bits (512K) real core address. The MMD works as follows. The 22-bit virtual address space is visualized as consisting of 4K pages, each 1K words long. A 22-bit address specifies a page number (a page table location) with its high order 12 bits, and a location within the page with its low order 10 bits. The main component of the MMD is a 4K by 16 bits random access memory that acts as a page table, i.e. it maps a virtual page into a real memory page. The 16-bit word at the specified location in the page table contains information on how to access the page. If the page is in core, its real core address, age, and status are specified as follows

AGE 5 bits - Clock reading at time page was last referenced (Modulo 32). AGE is jammed in from a special Age register every time the page is referenced. The Age register is incremented by software at presettable clock intervals.

WMOD 1 bit - Tells whether or not the page has been modified by a write access.

WPR 1 bit - Grants or refuses permission to write on the page.

RCP 9 bits - real core page number
(most significant 9 bits of 19 bit
Real core address)

If the page is not in core, AGE is set to a special value, and the Page Management module (see below) takes over.

The MMD hardware constructs a 19-bit address by concatenating RCP with the least significant 10 bits from the virtual address. An indication of the efficiency of performance we expect can be gleaned from the following example: To obtain CAR of a list (i.e. its first element) without the MMD would take 23 microseconds under the most favorable circumstances. Of these, 14 microseconds are used for software mapping one data reference; the other memory references are assumed non-mapped (which implies that the stack is windowed in core). With the MMD, this time is cut to 10.8 microseconds, and special care to keep the stack windowed is not required.

Mode registers

The page table provides the transformation from a 22 bit virtual address to a 19 bit real core address. The mode registers and the EXT register provide ways of specifying the 22 bit virtual address. The PDP-11 provides only a 16 bit (byte) address in the processor or an 18 bit (byte) address after memory management transformation.

We use the normal PDP-11 memory management in a rather unusualy manner to interface with the MMD. Memory management provides user and kernel mode, each with its own set of memory management registers (for two distinct 32K word address spaces). In addition it provides protection against halts in user mode. The input to memory management is a 16 bit address the output is an 18 bit address. We have used the two extra bits, not as address bits, but as new mode selection bits. The 2 bits select one of four mode registers. The mode registers then specify how the virtual address is to be created.

The EXT register is a 6 bit register that can be used to specify the high order 6 bits of an address. It can be set from micro-code from the D-bus by using a previously unused combination of one of the micro-code fields (SPS). The mode registers specify whether or not the EXT register is to be used for the high order bits of the address.

Since most INTERLISP data is stored in word or multiple word sized chunks, it would be wasteful of space to include an extra bit in each address for a byte address. And it would waste time in micro-code to shift a word address pointer left one to make it a byte address. Thus we have included a word addressing mode under control of the mode registers.

Another feature provided by the mode registers is the ability to do I/O to virtual addresses thru the MMD while LISP is running. The REL field in the mode register is an alternate means of specifying the high order portion of an address.

We outline briefly how the mode registers are used. One mode register is dedicated to LISP, one to the operating environment, and two to I/O. LISP runs in user mode and the user mode memory management registers are set up for a 1-1 mapping of the low order 16 bits of address and the high order 2 bits from memory management are always 01, selecting mode register 1. Mode register 1 is set to enable the page table, enable word addressing, and enable the EXT register.

The operating environment runs in kernel mode and the kernel mode memory management registers are set up for a 1-1 mapping of the 16 bit address in the range 0-24K, and the high order 2 bits select mode register 0. Mode register 0 is set for page table enabled, word addressing disabled, EXT disabled, and REL all ones so the operating environment uses the top most 32K chunk of the virtual address space. Addresses 24-32K in kernel mode are the so called I/O addresses and are selected by all ones in bits 17:14 of the output of memory management. This bypasses the MMD entirely.

Age Distribution Registers

The AGE field in each page table entry permits efficient demand paging by making it possible to find the least recently used pages. Such pages are good candidates for removal from memory when one needs to make room for new ones. Since the table is fairly large (4K words) a pure software search for the least recently used pages will be time consuming and inefficient; for this reason, we have included a set of age distribution registers.

These registers provide a histogram of the page ages existing in the table. In fact, there are 2 sets of age distribution registers, one for pages that have been modified, the other for unmodified pages. That is, for each possible combination of AGE and WMOD there is an Age Distribution register that contains the number of pages whose table entries contain that combination. Thus, for example, to find the n oldest pages one finds the corresponding ages by examining at most 64 Age Distribution Registers, and then uses these ages as search keys in scanning the map.

The complete set of specifications for the MMD can be found in Appendix B.

Software Implementation

The *software implementation of* INTERLISP is greatly facilitated by the fact that a large fraction of the system is written in LISP. The non-LISP parts of INTERLISP represent

1) the way INTERLISP is integrated with its operational environment,
2) the kernel, or irreducible parts that must be written in machine language and on which the rest of the system is built.

In the following we discuss these issues in detail.

Operating Environment

This package provides the facilities of a simple operating system, i.e.: the interfaces to specific hardware devices and the scheduling and performance of core allocation procedures.

This portion of the system is small in accord with our desire to minimize the machine dependence of the system.

Page management is the major component of the operating environment. The page manager allocates real core for the 4 million word LISP virtual address space using the memory mapping device for efficient demand paging. It uses 3 levels of storage: core, fixed head swapping disk, and bulk storage disk. The page manager also provides user mode access to the paging mechanism such as mapping file pages into address space, and get/set access permission for a page.

In addition, the operating environment supplies primitive file operations (read/write-character, open/close file); terminal i/o and interrupts; interfaces to IMP 11A, auto dialer, real time clock; and scheduling of background processes.

LISP Kernel

The LISP Kernel includes the following modules:

1) storage allocation and garbage collection;
2) data storage and retrieval;
3) stack management;
   (a) function call/return;
   (b) stack allocation and deallocation (note that multiple environments make this a fairly complex process), and
   (c) stack primitives - e.g. STKPOS, STKNTH;

4) the interpreter including EVAL, APPLY, ENVEVAL, ENVAPPLY, PROG, etc.;

5) other primitives - such as string and hash array manipulation

6) Basic I/O:

   (a) interfaces to the operating environment for page mapping, files, terminal, and other devices;

   (b) terminal interrupt handlers, and

   (c) atom hashing routine;

7) protection mechanisms for critical data such as file directories - Includes both protection against accidental destruction by user and protection against simultaneous access by asynchronous processes (e.g. user and FTP server);

8) compiled code interpreter.

The LISP kernel is implemented in a judicious combination of micro-code, PDP-11 code, and hand written L-code. Execution speed would be maximized if most of the LISP kernel were implemented in micro-code. However, it clearly will not fit in the available writable control store. Therefore, those portions of the kernel which are both frequently used and simple (short), such as the compiled code interpreter, function call/return and portions of the stack allocation and deallocation code are written in microcode. As for the other portions, while it is possible to swap microcode, the time to do the swap would generally exceed the time to accomplish the desired task with PDP-11 code or L-code instead.(1)

(1) An exception occurs with garbage collection. There are a

The rest of the LISP kernel that exceeds the capacity of the writable control store is written either in PDP-11 code or hand written L-code. At this level, it is clearly most desirable to minimize the amount of PDP-11 code because the L-code portions are directly transportable to different hardware. PDP-11 code is used primarily for initialization, and occasionally for operations that are not implementable or only awkwardly implementable in L-code. Perhaps surprisingly, at least 80% of the non-microcode portion of the LISP kernel is implemented in hand-written L-code. One might ask why we need hand-written L-code at all; why not write the desired functions in LISP and let the LISP compiler do the work? There are two reasons. First, there are some operations (e.g. APPLY), that can be expressed in L-code but not in LISP. Second, there is a bootstrap problem. In order to load functions compiled elsewhere the primitives to read the compiled file must be available. What remains to be explained is how to interface smoothly the execution of straight PDP-11 code, L-code, and LISP microcode, in particular how we intend to access the 4M word address space of the LISP environment while executing PDP-11 code, and how to get along with only 1000 words of WCS.

While it would be possible for PDP-11 code to access the 4M word LISP address space by using Core Management windows, this would be both slow and awkward. A better approach is to put to

---

small number of garbage collection primitives (those that mark and chase pointers) that are executed thousands or hundreds of thousands of times during a garbage collection. In this case it is feasible to swap microcode.

use some of the many unused PDP-11 opcodes and redefine them as new opcodes using the WCS. All these automatically cause a transfer of control to the writable control store, where one can decode the instruction, perform the desired operation in microcode (where access to the 4M word address space is faster and simpler, see EXT register in MMD appendix), and then resume PDP-11 code execution.

The reverse operation, i.e.: to transfer control from microcode to PDP-11 code is also necessary. To see this, just consider that the compiled code instruction set contains 2048 miscellaneous operations (see Appendix A). While the most frequently executed ones will be microcoded, it is clear that 1K of WCS can not handle the whole set. Therefore, most of these relatively infrequent compiled code instructions result in calls to routines in L-Code PDP-11 code. Another example is the set of operations involved in a return from a function. In the "spaghetti stack" (multiple environments) of INTERLISP there are two types of function returns, essentially the simple return and the return that requires an environment switch or environment copy. The simple return can be accomplished easily in microcode, while the hard return is probably both long enough and infrequent enough to be undeserving of microcode.

The hard return is handled in a manner similar to a PDP-11 trap. That is, we save the state of the machine, where the state includes whether we were in the compiled code interpreter or in 11-code (executing a "new" instruction - e.g. RETURN); we perform

the desired operation in 11 code; and we resume the interrupted state.

New LISP Code

As we noted before, a large portion of INTERLISP-10 is implemented in INTERLISP and may be transferred directly to INTERLISP-11 without modification. However, there is still a significant amount of additional LISP code that must be written for the PDP-11. The reasons are various -- some things are necessarily machine or implementation dependent s ch as the code generator for the compiler. Others are machine dependent (in INTERLISP-10) for speed, such as the arithmetic functions (SIN, COS, etc.), and READ and PRINT. Some things are provided by TENEX such as the file system.

INSTRUCTION SET

The LISP compiler will produce code in the instruction set described below. The code is interpreted (run) by the micro-code.

In the following description TOS means top of stack; the quantity on the top of the stack is referenced' S as a source means the quantity on the top of the stack is popped. S as a destination means the result is pushed on the stack.

Group 1

The group 1 operations have a 6 bit opcode and 10 bit source. The source is represented by a 7 or 8 bit offset from any of a number of bases.

The source types are:

000 STACK. Temporary value in the current frame extension. The 7 bit offset is a positive quantity representing a negative offset from the current stack pointer. (2*offset is # words)

001 LOCAL. Local variable in the current basic frame. The 7 bit offset is positive from current basic frame pointer. Offset is #words/2 = #stack slots.

010 SPEC. Specvar reference. An indirect reference to a value cell. In compiled functions the value cell pointers are in the literal area, and in interpreted functions the value cell pointers are in the basic frame. In either case the so-called literal pointer is the base for specvar references. The 7 bit offset is positive and = #words. (The value cell pointers are 1 word (16 bit) quantities; the high six bits are implicit.)

011 PVAR. An indirect reference to local variable of an outer frame, referenced from within a PROG or open LAMBDA that makes a frame. The base for the reference is the literal pointer. The 7 bit offset is # words. The PVAR literals are one word quantities containing a frame count and an offset.

01X LIT. Literal of the (compiled) function now running. The 8 bit offset is a positive offset from the current function literal pointer. Offset is #words/2. (All normal literals are 2 word quantities.)

110 SYSTEM CONSTANT. Constants that are referenced frequently, such as T and NIL, are stored in a system literal table. The 7 bit offset is positive from the beginning of the system literal table. (Offset is #words/2.)

111 IMMEDIATE. Immediate small numbers in the range -64 to +63.
The offset is the number & 177.

CAR and CDR

op-code	mnemonic	description
040XXX	PUSH	E->S
042XXX	RET	RET(E)
044XXX	unused	
046XXX	unused	
050XXX	CAR	CAR(E)->S
052XXX	RCAR	(return (CAR E))
054XXX	CDR	CDR(E)->S
056XXX	RCDR	etc.
060XXX	CAAR	
062XXX	RCAAR	
064XXX	CADR	
066XXX	RCADR	
070XXX	CDAR	
072XXX	RCDAR	
074XXX	CDDR	
076XXX	RCDDR	

Integer arithmetic

100XXX	IADD	E+TOS->TOS
102XXX	ISUB	E-TOS->TOS
104XXX	IMUL	E*TOS->TOS

106XXX	IDIV	E/TOS->TOS
110XXX	IREM	remainder(E/TOS)->TOS
112XXX	EQ	Compare TOS with E and POP.
		Set indicator TRUE if EQ.
114XXX	IGT	Set indicator TRUE if E>TOS.
		POP always.
116XXX	ILS	Set indicator TRUE if E<TOS.
		POP always.

Note that the quantity on the top of the stack can be either a pointer to a number (a boxed number) or a tagged unboxed number. The result is stored on the stack as a tagged unboxed number. Eventually we expect that the microcode will allow unboxed numbers on the stack in any situation and will check and box prior to binding, storing in list cell etc. However, for the moment it is the responsibility of the compiler to see that numbers get boxed before being used by non-numeric operations. (But the compiler does not have to be concerned with unboxing.)

General arithmetic

120XXX	ADD
122XXX	SUB
124XXX	MUL
126XXX	DIV
130XXX	REM
132XXX	EQP
134XXX	>
136XXX	<

Type tests

020XXX	LISTP	test E, set indicator.
022XXX	ATOM	
024XXX	LITATOM	
026XXX	NUMBERP	
030XXX	FIXP	
032XXX	STRINGP	
034XXX	ARRAYP	
036XXX	STACKP	

Group 2 - the stores

The "store" operations have a 7 bit op-code and a 9 bit destination. The destination is represented by a 2 bit destination type and a 7 bit offset. The destination types are a subset of the source types. (Since it doesn't make sense to "store" to a constant.) The destination types are: 00 STACK, 01 LOCAL, 10 SPEC, 11 PVAR.

010XXX	SCDR	CDR(E)->E
011XXX	POP	S->E
012XXX	SCDDR	CDDR(E)->E
013XXX	TOS	TOS->E
014XXX	ADD1	E+1->E

015XXX	PCAR	CAR(S)->E
016XXX	SUB1	E-1->E
017XXX	PCDR	CDR(S)->E

Note: POP, PCAR, and PCDR compute the effective address after they pop - so pops to stack can be a bit confusing.

Group 3 - Branches

The branches have a 6 bit op-code and a 10 bit offset allowing branches within -512 and +511 of the current location. An offset of 0 implies a long branch with a 16 bit offset in the following word.

140XXX	BRT branch if indicator true
142XXX	BRF branch if indicator false
144XXX	PBNIL POP and branch if TOS=NIL, else don't POP
146XXX	BR unconditional branch
150XXX	BNIL branch if TOS=NIL and POP always
152XXX	BNN branch if TOS NOT NIL and POP always
154XXX	NBNIL if TOS=NIL, branch and don't pop; if TOS NOT NIL, don't branch and do pop.
156XXX	NBNN if TOS NOT NIL, branch and don't pop; if TOS=NIL, don't branch and do pop.

Group 4 - Literal references

These operations have an 8 bit op-code and an 8 bit offset. The only operand type is the literal, and as with all literal references, the offset is a positive offset from the current literal pointer (#words/2).

0020XX	CALL	Call function, literal is # args., fnname

0024XX	DCALL	Call function, discard value
0030XX	LCALL	linked call to function
0034XX	DLCALL	linked call, discard value
0040XX	PBIND	Arg (immediate) is binding depth of prog - binds the PROG frame and gets the back pointers.
0044XX	unused	
0050XX	PCALL	Used for prog's and open lambdas that make frames. LITERAL loc is the beginning of an FEF for the frame.
0054XX	EQQ	EQ QUOTE - compares TOS with literal. POPs if EQ, else doesn't POP; sets indicator always. Used for Selectq.
0060XX	TYPTOS	Test type of TOS. Arg. is type number (immediate). If type is eq, leaves arg on TOS. If not eq, puts NIL on TCS. Has variant (0062) DTYPTOS that POPs. Sets indicator in any case.
0064XX		unused
0070XX		unused
0074XX		unused

Group 5 - Miscellaneous opcodes 0-001777

1024 miscellaneous operations. The arguments, if any, are on the stack. The arguments are generally popped, and the value, if any, is returned on the stack. Some of the miscellaneous opcodes are primitive operations necessary for compiled code. Others implement frequently called Lisp functions so that they may be compiled open to avoid function calls, and others implement primitive portions of more general Lisp functions so that the general functions can be written in Lisp.

The opcodes 0-377 are implemented entirely in microcode. Some have a parameter encoded in the 3 low order bits.

10	EXCH	Exchange the top two items on the stack
20	IBOX(N)	Box the integer on TOS, result to TOS
30	GCONS(TYP)	General cons of specified type (a type number). Returns the next free cell.
40	CONS(A D)	
50	BIND	Binds the current basic frame. This is the first instruction in any compiled function that binds any SPECVARs. This is a separate operation from the function call in order to satisfy the requirement that all instructions be restartable.
60-67	DPOPN	Pop and discard 0-7 (encoded in low 3 bits) items from the stack.
70-77	IRET	Internal return. Used to return from those miscellaneous opcodes that are implemented in L-code. The item on TOS is returned as value and 0-7 (encoded in low 3 bits) prior items are popped.
100	BIN(IOD)	Read a character from IOD or string. Skip if successful, else don't skip.
110	BOUT(IOD CHAR)	Write a character to IOD or string. Skip if successful, don't skip if full.

120	RPLACA (L X)	Replace CDR of L by X. (Note: does not check that L is a list.) Value is L.
130	RPLACD(L X)	Replace CDR of L by X. (Note: does not check that L is a list.) Value is L.
140	DRPLACA (L X)	RPLACA and discard value.
150	DRPLACD (L X)	RPLACD and discard value.
160-167	SCALL(A1...AN	FN) Call function FN with arguments Ai. Number of arguments supplied is encoded in the low 3 bits.
170	LLM	Leave LISP mode at current PC.
200	R16I(HEAD N)	Read 16 bits from the Nth word of the structure HEAD. Value is an integer.
201	R16S(HEAD N)	Read 16 bits from the Nth word of the structure HEAD. Value is an unboxed stack pointer.
202	R16V(HEAD N)	Read 16 bits from the Nth word of the structure HEAD. Value is a value cell.
210	W16(HEAD N VAL)	Write 16 bits in the Nth word of the structure HEAD. The low

16 bits of VAL are written (with no type checks).

220 RPTR1(HEAD N) Read pointer from the Nth word of the structure HEAD.

230 WPTR1(HEAD N VAL) Write pointer VAL in the Nth word of the structure HEAD. Value is HE D.

240 RPTR2(HEAD N) Read the second half of the packed pointer pair beginning at word N in the structure HEAD.

250 WPTR2(HEAD N VAL) Write VAL in the second half of the packed pointer pair beginning at word N in the structure HEAD. Value is HEAD.

260 RNUM(HEAD N) Read 24 bit integer from NIL word of the structure HEAD. Value is a boxed integer.

270 WNUM(HEAD N VAL) Unbox VAL and write the resulting 24 bit integer in the Nth word of the structure HEAD. Value is HEAD.

300	IVAL	Push the value of the indicator; i.e. push T if the indicator is true, NIL if false.
310-317	PBIND	Bind the basic frame for a PROG or open LAMBDA. Also store in the frame extension the pointers to the frame extension and basic frame (16 bit pointers in that order) of each outer PROG or LAMBDA and the main function. The binding depth is in the low 3 bits.
320	LOGOR (x y)	Logical OR of 2 arguments
330	LOGAND(X y)	Logical AND of 2 arguments
340	LOGXOR(x y)	Logical XOR of 2 arguments
350	LSH(x n)	Shift x left N bits if N positive, right |N| if N negative.
360	EXFRM	Make frame for EXPR. Expects callers PC and a 2 word quantity containing the number of arguments stacked from the EXPR and the size of the FEF. (1)

370 DIVAL Value of the indicator to TOS.

The miscellaneous opcodes 400-1777 are all implemented in hand code; that is, in hand written L-code with a small amount of PDP-11 code where necessary. The micro-code basically executes a PUSH through a dispatch table in resident main memory; IRET is used for return.

(1) The process of calling an EXPR is fairly involved due in part to the shortage of micro-code space and in part to the requirement that instructions be restartable. A function call operation such as CALL or SCALL, when it detects an EXPR, will go out to L-code to adjust the number of arguments for the EXPR and to create an FEF for the call. Then EXFRM is executed from L-code to create the frame. EXFRM then exits to L-code again to actually evaluate the EXPR.

400	MEMB(X L)	
401	ASSOC(X L)	
402	A.ATOM(PNAM)	Allocate new atom
403	A.PNAM(N)	Allocate space for pname
404	GETD(ATOM)	Returns a fncell if definition is compiled, else returns e.g. a list. If arg not atom, returns NIL.
405	PUTD(atom def)	Atom must be a LITATOM. Def may be fncell, litatom, or list. If litatom, the definition is put, so can do MOVD without making a fncell.
406	GETPROPLIST(ATOM)	
407	SETROPLIST(ATOM)	
410	A.ARRAY(N TYP)	Allocate array with room for N elements, TYP=0 pointer array. TYP=1 integer array. (TYP=3 floating point & TYP=4 hash array not implemented yet.)
411	A.STPT()	Allocate empty string pointer
412	A.DSTR(N)	Allocate dummy (garbage) string of N characters.
413	L.SBST(S1 N M S2)	Primitive substring. S1 and S2 must be string ptrs, N and M must be integers. M may be negative. N must be positive. S2 is a string ptr to re-use and is the value.
414	L.SBSN(S1 N M S2)	Ditto to above but doesn't check S2. Used by IODSPTR.
415	LIST(A1 a2..N)	List of indefinite number or args.
416	EVALV(atom)	Evaluate atom in current environment - value is NOBIND if unbound.
417	GTVC(atom)	Get value cell of atom.
420	VCTOAT(vcell)	Given value cell, gets atom.
421	MKAT(string)	Makes a LITATOM given string as its PNAME. Does no parsing. Uses the whole string.
422	ELT(array N)	Array must be an array, N must be a number in the range of the array. Sorts out array type and boxes value if integer array. (For hash array ELT will be a list cell of KEY VAL)
423	SETA(array N val)	Checks same as for ELT. Also val must be integer if array type is integer.
424	L.CHCN(atom/string)	Primitive CHCON1. Returns character code of first character of LITATOM or STRING.
425	SBIN(string)	Gets first character (code) of string. Steps string pointer. Return NIL if string is empty.
426	SBOUT(char string)	Writes character (code) into first character of string. Steps string pointer. Returns NIL if no more string. SBOUT destroys both the string and the string pointer. (Can clobber

pnames.)

427	OBIN(IOD)	Get next character (code) from.IOD. Call FN specified in IOD when no more characters. If FN is NIL, returns NIL when empty.
430	OBOUT(char IOD)	Write next character to place specified by IOD. Call FN as above.
431	TTYIN()	Get character from terminal (no echo).
432	TYOUT()	Write character to terminal.
433	CHARACTER (code)	Returns the atom whose pname is the character specified by CODE.
434	SET(atom val)	
435	A.CODE(N)	Allocate space for compiled code.
436	CCODEP(ATOM/DEF)	Predicate. T if given an atom whose definition is compiled or if given a function cell containing a compiled definition. NIL otherwise.
437	GETDP(atom)	Predicate. T if arg is an atom with non-NIL function definition, NIL otherwise.
440	TRATM()	Temporary. RATOM from TTY.
441	L.NCHR(atom/string)	Primitive NCHARS. Returns number of characters in atom or string.
442	ATTOST(atom STR)	Makes string cut of PNAME of atom. Reuses string ptr.
443	GETIOD(string FN)	Create IOD for string. FN is name of function to call when string is exhausted. If reading, FN is called with 1 arg, the IOD. If writing, FN is called with 2 args, the char and the IOD. (To get IOD for a file use OPEN - not implemented yet.)
444	IODGNC(IOD)	Get "next char" field from IOD. (Put there by READ, RATOM, etc.)
445	IODSNC(IOD CHAR)	Set "next char" field.
446	IODGLC(IOD)	Get "last char" field from IOD.
447	IODSLC(IOD CHAR)	Set "Last char" field in IOD.
450	IODGPOS(IOD)	Get position field of IOD. Number of characters since last line feed.
451	IODSPOS(IOD N)	Set position field in IOD.
452	IODGPTR(IOD)	Get pointer field from IOD. i.e. number of characters written/read so far.
453	IODSPTR(IOD N)	Set pointer field in IOD.
454	IODGBAS(IOD)	Get the base field of IOD - i.e. original string or file description.
455	SRATM(IOD)	Primitive version of RATOM from IOD or string. Used by LAPRD.
456	LAPRD(IOD FN)	Read compiled definition of FN from IOD or string.
457	ARRSIZ(ARRAY)	Returns the number

		of elements in ARRAY.
460	EXENT	Used in the process of calling an EXPR.
461	EXEV	Used to evaluate an EXPR
462	PROGN(E1...EN)	Implements PROGN.
463	L.APP.	Implements APPLY*. The definition of APPLY* is (LAMBDA A (L.APP.)).
464	L.APPLY(FN ARGS)	Implements APPLY.
465	L.EVF(FORM)	Implements EVAL of simple form. The rest of EVAL is implemented in Lisp.
466	ARGTYP(FN)	Determines the argument type of FN. FN may be a litatom, fncell, or EXPR. The value returned is a number interpreted as follows: 0 LAMBDA 1 NLAMBDA 2 LAMBDA no spread 3 NLAMBDA no spread Value is NIL if FN is not a legal function.
467	NTYP(X)	Returns the numerical data type of x.
470	NULL(X)	Implements the function NULL.
471	L.PROG(ARGS E1..EN)	Implements interpreted PROG - the definition of PROG is (NLAMBDA A (L.PROG A)).
472	L.GO(LABEL)	Implements interpreted GO
473	unused	
474	LAST(L)	Implements the function LAST.
475	LENGTH(L)	Implements the function LENGTH
476	NTH(L N)	Implements the function NTH
477	LOAD(address)	Bootstrap version of LOAD that leads a compiled file from a buffer in low core.
500	L.CSP(old new)	Copy string pointer from OLD to NEW. Value is NEW.
501	L.DDT	Enter user DDT

Included below are the LISP functions REVERSE and COPY, the L-code for the functions, and for comparison the PDP-10 code for the same functions.

```
(REVERSE
  (LAMBDA (L)
    (PROG (U)
          (DECLARE (LOCALVARS U))
      LOOP(COND

       ((NLISTP L)

        (RETURN U)))
          (SETQ U (CONS (CAR L)

                        U))
          (SETQ L (CDR L))
          (GO LOOP))))

(COPY
  (LAMBDA (X)
    (DECLARE (LOCALVARS Y Z))
    (COND
      ((NLISTP X)

  X)
       (T (PROG (Y Z)

          (SETQ Y (SETQ Z (LIST (COPY (CAR X)))))

      LP  (COND

              ((NLISTP (SETQ X (CDR X)))

                (FRPLACD Z X)

                (RETURN Y)))

          (SETQ Z (CDR (FRPLACD Z (CONS (COPY (CAR X))))))

          (GO LP))))))
```

PDP-11 L-CODE

```
REVERSE
        (BIND)
        (PUSH (LAPLIT NIL))
LOOP    (LISTP (VREF L 1))
        (BRT (TREF 4))
        (RET (VREF U 1))
4       (CAR (VREF L 1))
        (PUSH (VREF U 2))
        (CONS)
        (POP (VREF U 1))
        (SCDR (VREF L 1))
        (BRA (TREF LOOP))
FEFORG (1 . 0)
       L
```

13 WORDS, 208 BITS

```
COPY
        (BIND)
        (LISTP (VREF X 0))
        (BRT (TREF 2))
        (RET (VREF X 0))
2       (PUSH (LAPLIT NIL))
        (PUSH (LAPLIT NIL))
        (CAR (VREF X 2))
        (CALL (LAPLIT COPY 1))
        (CONSNL)
        (TOS (VREF Z 3))
        (POP (VREF Y 2))
LP      (SCDR (VREF X 2))
        (LISTP (VREF X 2))
        (BRT (TREF 6))
        (PUSH (VREF Z 2))
        (PUSH (VREF X 3))
        (DRPLACD)
        (RET (VREF Y 2))
6       (PUSH (VREF Z 2))
        (CAR (VREF X 3))
        (CALL (LAPLIT COPY 1))
        (CONSNL)
        (RPLACD)
        (PCDR (VREF Z 2))
        (BRA (TREF LP))
FEFORG (1 . 0)
       X
LITORG (1 . COPY)
```

29 WORDS, 464 BITS

```
PDP-10 COMPILED CODE

REVERSE
        (JSP 7 , ENTERF)
        (262144 0)
        (0 PLITORG)
        (PUSH PP , ' NIL)
LOOP    (HRRZ 1 , (VREF L 1))
        (JSP 6 , SKNLST)
        (JRST (TREF 4))
        (HRRZ 1 , (VREF U 1))
        (RET)
4       (HRRZ 1 , @ (VREF L 1))
        (HRRZ 2 , (VREF U 1))
        (PUSHJ CP , CONS)
        (HRRM 1 , (VREF U 1))
        (HLRZ 1 , @ (VREF L 1))
        (HRRM 1 , (VREF L 1))
        (JRST (TREF LOOP))
LITORG
PLITORG (VALUE-CELL L)

                                17 WORDS, 612 BITS

COPY
        (JSP 7 , ENTERF)
        (262144 0)
        (0 PLITORG)
        (HRRZ 1 , (VREF X 0))
        (JSP 6 , SKNLST)
        (JRST (TREF 2))
        (POPJ CP ,)
2       (PUSH PP , ' NIL)
        (PUSH PP , ' NIL)
        (HRRZ 1 , @ (VREF X 2))
        (ACCALL 1 , ' COPY)
        (PUSHJ CP , CONSNL)
        (HRRM 1 , (VREF Z 2))
        (HRRM 1 , (VREF Y 2))
LP      (HLRZ 1 , @ (VREF X 2))
        (HRRM 1 , (VREF X 2))
        (JSP 6 , SKNLST)
        (JRST (TREF 6))
        (HRRZ 1 , (VREF Z 2))
        (HRRZ 2 , (VREF X 2))
        (HRLM 2 , 0 (1))
        (HRRZ 1 , (VREF Y 2))
        (RET)
6       (PUSH PP , (VREF Z 2))
        (HRRZ 1 , @ (VREF X 3))
        (ACCALL 1 , ' COPY)
        (PUSHJ CP , CONSNL)
        (MOVE 2 , 1)
```

```
        (POP PP , 1)
        (HRLM 2 , 0 (1))
        (HLRZ 1 , 0 (1))
        (HRRM 1 , (VREF Z 2))
        (JRST (TREF LP))
LITORG
PLITORG (VALUE-CELL X)
        COPY
```

35 WORDS, 1260 BITS

Storage allocation

As in Interlisp-10, the data type of a pointer is determined by the address; that is, storage is allocated in pages and the data type of the page is stored in a table. In Interlisp-11 we also have a larger unit of allocation, the segment. A segment is 64K words (64 pages). In order to save space, some data types are restricted to one preallocated segment so that pointers to them can be 16 bits in contexts where the type is known. This is particularly useful in stack frames that contain pointers to other locations on the stack (ALINK, CLINK, and BLINK).

Currently there are preallocated uniform segments for stack, value cells, and small integers in the range -32K to +32K-1.

Modes, machine state, and additional instructions

The operating environment runs in PDP-11 kernel mode; LISP runs in PDP-11 user mode. While the part of Lisp written in Lisp is entirely in L-code, the Lisp kernel is implemented partially in L-code and partially in PDP-11 code. Thus, when in user mode, part of the machine state is whether we are running L-code or PDP-11 code. We refer to these as Lisp mode and 11 mode.

We have implemented in micro-code some additions to the PDP-11 instruction set. These new instructions serve a variety of purposes.

1) To save and restore the machine state. These instructions are used by the operating environment to save state when a trap occurs (e.g. page fault) and to resume after the trap is processed. They are necessary both for efficiency and because some of the state is not directly accessible from PDP-11 code.

2) To allow the operating environment to access the entire virtual address space without using windows.

3) To allow the access from 11-mode code in the Lisp kernel to the Lisp environment - e.g. the virtual address space and stack.

Before describing the additional instructions we will briefly discuss how the PDP-11 general registers and other state registers are used by Lisp.

The PDP 11/40 has 16 general registers. Registers 0-7 are accessible from PDP-11 code. Register 16 is user mode register 6. The other 7 general registers are accessible only from micro-code. We have used some of these "hidden" registers to hold status information for Lisp-mode code.

The general register assignments are:

R0,1	temporary pointer
R2	temporary 16 bit quantity
R3	VP, pointer to current basic frame

R4 NPP, number of slots remaining on the stack

RR5 PP, pointer to top of stack

R6 kernel stack pointer

R7 PC, current program counter
(low 16 bits)

R15 PCHI, bits 5:0 are the high order
6 bits of the program counter.
Bit 6 is a mode indicator; 0 if in
11 mode, 1 if in Lisp mode.
Bit 15 is the 2 word instruction indicator;
1 is normal, 0 means a 2 word instruction is
in progress.
Bits 14:7 are the high order 6 bits of the
literal or FEF pointer.

R16 LIT. If the currently running function is
compiled, LIT is the low order 16 bits of the
pointer to the literal area of the function.
If the current function is interpreted, LIT
points to the FEF for the current frame.
(Note that R16 is user mode R6.)

R17 PPINC, keeps track of the number of times PP,
the Lisp stack pointer, has been incremented in the
current instruction so that the stack pointer can
be backed up if a page fault occurs in the middle
of an instruction.

Other status information:

PS — Program status, has regular PDP-11 usage except PS bit 1 is used for the true false indicator in Lisp mode.

MR1 — mode register 1. This is the mode register used by Lisp. In 11-mode it is set for byte addressing, extension register disabled.
In Lisp mode it is set for word addressing, extension register enabled.

New PDP-11 instructions

210 TRPSAV

This instruction only works in kernel mode. It is used to save the machine state prior to processing a trap. Saves R0-R7, PS, PCHI, user R6, MR1, and EXT registers, in that order, in the block pointed to by CSAVPB (byte address 376). If the trap occurred from kernel mode, saved PCHI is always 0, and MR1 and EXT are not saved. If the trap occurred from user mode, and MR1 has word addressing enabled, Lisp mode is implied. In this case, the stack pointer is adjusted by PPINC and PCHI is saved. If MR1 has word addressing disabled then 11-mode is implied and saved PCHI=0.

211 TRPRES

TRPRES only works in kernel mode and is used to restore the machine state after processing a trap. Restores the state from two blocks; an "old" block addressed by R0, and a "new" block addressed by R1. (R0 and R1 may be equal.) The contents of R1 are stored in CSAVPB.

Restores PC (R7) and PCHI from the "new" block. Restores R0-R5, user R6, and PS from the "old" block. If PS implies kernel mode then sets kernel R6 from the "old" block. If PS implies user mode, then sets MR1 and on the basis of PCHI resumes either 11-code or L-code.

212 RSETK

This instruction is used in only one place. In the current configuration of WCS and memory management it is not possible in micro-code to go from user to kernel mode by simply clearing the appropriate bits in the PS. So to get into kernel mode we fake a trap to the vector at location 374 which is set up to enter kernel mode and execute RSETK. RSETK just resumes the microcode at the microcode location saved on the micro stack.

213 ECONS

Cons (S TOS) => TOS

214 RR2

Load R2 with the 1 word quantity addressed by the 22 bit virtual address in R0, 1.

215 WR2

Store R2 in the location addressed by the 22 bit virtual address in R0, 1.

216 ECAR

CAR (R0,1) => R0,1

217 ECDR

CDR (R0,1) => R0, 1

220 EPUSH

Push R0,1 to Lisp stack.

221 EPOP

Pop Lisp stack to R0,1.

222 EDPOP

Pop Lisp stack, discard value.

223 ELM

Enter Lisp mode at current PC.

224 FET

Fetch to R0,1 the 32 bit quantity addressed by the pointer on TOS.
The stack is not popped.

225 STO

Store R0,1 in the two words beginning at the location addressed by the pointer on TOS.
The stack is not popped.

226 EEXCH

Exchange R0,1 and TOS.

227 BLT

Block transfer. R0,1 has the destination, R2 has the number of words and TOS has the last source.

70XX RSTK

Load R0,1 with the contents of the Nth slot from the top of the stack.
N is in the low 6 bits of the instruction (destroys R2).

71XX WSTK

Store R0,1 in the Nth slot from the top of the stack. N is in the low 6 bits of the instruction (destroys R2).

Data formats

Stack format

The stack is allocated in a fixed 64K segment, so that a stack pointer can be specified in 16 bits. The Interlisp "spaghetti" stack is composed of linked stack frames with separate links for control and variable access. A frame is subdivided into the basic frame which contains the variable bindings, and the frame extension which contains the links, control information, and temporary values. Several frame extensions may share the same basic frame. Both the frame extension and the basic frame contain reference counts. Since the stack may become fragmented in the course of computation, we must also have a way to keep track of unused stack space. Thus, there is also a special format for a stack hole. Note that the formats have been chosen so that each item occupies an even number of words.

Frame Extension

word 0	bit 15	0 active, 1 unactive
	bit 14	0 easy return, 1 hard return
	bits 7:0	use count
word 1	END-1	pointer to last stack slot in frame extension (or to the last word minus 1)
word 2	BLINK	pointer to basic frame
word 3	ALINK	pointer to next frame back in access chain
word 4	CLINK	pointer to next frame back in control chain
word 5	bits 13:8	high order 6 bits of literal pointer
	bit 7	0 normal, 1 discard value on return to this frame
	bit 6	mode bit; 0 PDP-11 mode, 1 Lisp mode
	bit 5:0	high order 6 bits of

		return address
		(resumption point on return
		to this frame)
word 6	RETLO	Low order 16 bits of
		return address
word 7	LITLO	Low order 16 bits of
		literal pointer for
		this frame.

The remainder of the frame extension contains an arbitrary number of temporaries each occupying a 2 word slot (32 bits). Since a pointer requires only 22 bits, the extra bits in a stack slot can be used for a variety of magic markers. In particular, 24 bit unboxed integers can be stored on the stack.

Basic frame

The fixed overhead of the basic frame is stored at the end of the frame following the bindings. This allows us to stack the arguments to a function and then to create the basic frame without having to move the arguments. The BLINK of the frame extension points to the beginning of the fixed overhead. The beginning of the basic frame contains an arbitrary number of variable bindings each occupying a two word slot (32 bits). In the case of a local variable(1) , the binding slot simply contains the variable value. In the case of a special variable or specvar(2) , the situation is more complicated. The "names" of specvars are found in the FEF (function entry frame) of the frame. The FEF is in the literal area in the case of a compiled function and in the frame extension in the case of an interpreted function. Now since we use shallow binding, the current value of a specvar is in the value cell of the atom name, and the previous value is in the basic frame; binding is the process of putting the new values in the value cells and the old values in the basic frame. This is the situation when a basic frame is part of the active computation. However, because of the "spaghetti" stack, a basic frame can also be inactive, in which case the basic frame contains the value as of that frame. In Interlisp-10 there is a single bit in the basic frame that tells whether the basic frame is active (bound) or inactive (unbound). But in Interlisp-11 there is an additional complication. The process of binding a frame is done in a single L-code instruction, with the potential for referencing several

(1) local variables are anonymous, and inaccessible from other frames.
(2) specvars are global, and accessible to lower level frames.

pages any of which may cause a page fault. Thus, since we wish all instructions to be re-executable if a page fault occurs during execution, there is a bit in each binding slot that tells whether the variable is in the bound or unbound state.

The basic frame overhead is:

word 0	bits 7:0	CXT - count of extensions that use this basic frame
word 1	BEG-2	pointer to first binding - 2. This is the value for VP, the variable pointer, when the frame is active.
word 2	bits 15:8	number of bindings
	bits 5:0	high order part of frame name
word 3	NAMELO	low order part of frame name.

Holes

word 0	-1	This is the hole flag
word 1	END	pointer to end of hole
word 2	LAST	pointer to previous hole
word 3	NEXT	pointer to next hole.

Stack holes are chained both frontwards and backwards, permitting quick searches for available holes, and permitting removal of any hole from the chain.

When a frame is about to be run, it is desirable to be able to determine whether a stack hole immediately follows the extension. Thus, the hole mark must be distinguishable from any binding and from a frame extension header. Thus, we prohibit all ones in the high order 10 bits of a binding, and all ones as the value of USE.

A two word hole cannot be included in the hole chains, but can be used if required by the frame directly above or can be merged with adjacent holes if and when they occur.

Size Comparison

Extension overhead	PDP-11	PDP-10
Basic frame overhead	64 bits	36bits
compiled bindings	32 bits	36 bits
interpreted bindings	48 bits	36 bits
Total (3 args compiled)	288 bits	324 bits
total (3 args interpreted)	336 bits	324 bits

Atoms

A literal atom has four components: pname (print name), function cell, value cell, and property list. The components are stored in a 6 word datum as follows:

word 0	function cell	
word 1	function cell	
word 2	pointer to value cell	
word 3	bits 13:8	high order 6 bits of property list
	bits 5:0	high order 6 bits of pname
word 4		low order 16 bits of pname
word 5		low order 16 bits of property list

The function cell is a 32 bit quantity containing the following

bits 31:30	function type (lambda, nlambda, spread, no-spread)
bits 29:23	number of arguments
bit 22	0 if expr, 1 if compiled
bits 21:0	pointer to the function definition

Note that bits 31:23 are only meaningful for compiled functions. For exprs this information must be obtained from the function definition.

There is also a separate data type called a function cell which contains the same 32 bit encoding of a function definition. A function cell is returned as the value of GETD of a compiled function, and may be used in place of a function name or Lambda expression when calling EVAL or APPLY.

The value cell contains the current binding of the atom. The actual value cells are allocated in a fixed 64K segment. What is contained in the atom is a 16 bit pointer to the relative location of the value cell in the segment. In addition to the current binding, the value cell contains a pointer back to the atom. The pointer back to the atom is required for backtrace, STKARGNAME, and other functions that need the name of a binding. The value cell format is:

word 0	bits 13:8	high order 6 bits of atom
	bits 5:0	high order 6 bits of value
word 1		low order 16 bits of value
word 2		low order 16 bits of value

The reasons for this method of storing values are:

1) It saves space in compiled code since the value cells can be referenced directly with a 16 bit pointer.
2) Preliminary studies have shown that only 25% of the atoms in a "typical" Interlisp system have value cells. Thus clustering the value cells in a single segment should help to reduce page faults.
3) It results in an overall space saving since an additional 16 bits would be required in the atom to contain the full 22 bit value, while the extra three words for the value cell use an average of .25x48, or 12 bits per atom.

The average space required for an atom is 108 bits. In Interlisp-10 the average space is 117 bits.

Compiled Code

A compiled code block contains a 4 word header giving the positions of various parts, followed by the L-code for the function, then the FEF's (function entry frames), next value cell pointers for variables referenced freely, and last the literals (constants) used in the function.

The header format is:

word 0	length, maximum 64K words
word 1	FEF beginning relative to word 0
word 2	bits 15:8 number of words in FEF's
	bits 7:0 number of value cell pointers
word 3	literal beginning relative to word 0

The first FEF is for the function, and it may be followed by FEF's for internal PROGS and open LAMBDAs. For each SPECVAR bound in the function the FEF contains an 8 bit argument number and a 16 bit value cell pointer arranged as follows:

word 0	bits 15:8 argument #
	bits 7:0 argument #
word 1	value cell pointer
word 2	value cell pointer
.	
.	
.	
etc.	

The FEF is terminated by an argument number equal to 0. The FEFs are followed by the 16 bit value cell pointers for variables referenced freely by the function.

Next are the "prog variable indicators" or PVIs. These are used to reference local variables of outer PROGs or the main function from within a PROG that makes a frame. The left byte of the indicator is the relative location of the desired frame pointer in the current frame extension, and the right byte is the position of the variable within the frame.

The normal literals are 2 word quantities containing 22 bit pointers and possibly other information. Specifically, the literal for a function call contains the number of arguments supplied in the high order bits.

We use a single base register for references to all this information in the compiled function. This, combined with variable reference offsets of 7 bits, literal reference offsets of 8 bits, and 2 word literals, imposes the following rather peculiar limits.

1) number of FEF words plus number of vcells plus number of PVIs must be less than or equal 128 words, and must be an even number of words (filled with a 0 if necessary)
2) (# FEF plus #PVI plus #VCP)/2 plus number of literals must be less than or equal 256.

A further limitation is that a compiled code block cannot overlap a segment (64K) boundary. In this way we avoid a double precision add to increment the PC (If this becomes a serious limitation we could have a special branch instruction for crossing segment boundaries).

List Storage

A list cell can occupy one, two or three 16 bit words. The chosen encoding is not the most compact encoding possible but it is quite compact while remaining fairly simple to encode and decode.

The high order 3 bits of each cell of whatever size contain the type. The types are

#	TYPE	FORMAT	
000	SHORT	bits 15:13	TYPE
		bits 12:6	7 bit CAR
		bits 5:0	6 bit CDR
001	SHORT CAR	bits 31:29	type
		bits 28:22	7 bit CAR
		bits 21:0	full CDR pointer
010	SHORT CDR	bits 31:29	type

		bits 28	unused
		bits 27:22	6 bit CDR
		bits 21:0	full CAR pointer
011	LONG	bits 47:45	type
		bit 44	unused
		bits 43:38	high order 6 bits of CAR
		bits 37:32	high order of 6 bits CDR
		bits 31:16	low order 16 bits CAR
		bits 15:0	low order 16 bits CDR
100	INDIRECT-1	bits 15:13	type
		bits 12:0	offset from this location of actual list cell
101	INDIRECT-2	bits 31:29	type
		bits 28:22	unused
		bits 21:0	pointer to actual list cell

The 7 bit short CAR is encoded as follows. If all 0, then CAR is NIL. Otherwise, if the high order bit is 0, then CAR is a list, and the remaining 6 bits are the offset from the location in words. If the high order bit is 1, then CAR is a small integer between -32 and +31.

The 6 bit short CDR is coded similarly; that is, 0 means CDR is NIL, anything else means CDR is a list offset by +/-31 words.

Indirect types are used when an RPLACA or RPLACD occurs to a short cell, where the result will not fit in the cell. An offset of 0 in type INDIRECT-1 means the actual cell is in a hash table keyed on the original location.

Using the following data obtained by Green and Clark we can compute the average number of bits required by a list cell.

a) 29% of CAR's are lists. Of these 70% are within +/-31 cells.
b) 72% of CDR's are lists. Of these 80% are within +/-31 cells
c) 25% of CDRs are NIL
d) 3% of CARs are NIL

e) 11% of CARs are small integers 0-15

Thus if we assume no indirect pointers, the average number of bits per list cell is 28.5 bits.

(Note that we have assume an equivalence of cells and words which is incorrect. However, adjusting for this does not change the results significantly).

---

*This list cell encoding has not been implemented yet.

Working set size

In our Interlisp-10 work, we obtained some data on desirable working set sizes for large Interlisp-10 programs, and the distribution of the various data types in the working set. Briefly 100-150 TENEX pages is adequate for most programs. A page in TENEX is 512 36 bit words. The distribution by data type does not vary significantly between 100 and 150 page working sets, so we give the data for the 100 page working set only, averaged over 3 programs

data type	pgsinworking set-TENEX	11/10 ratio(bits)	PDP-11words
MACROCODE	19	.5 (est.)	10944
COMPILED	34	.37	14492
ARRAY	2.75	.89	2819
STACK	4	.89	4101
LIST	13	.81	12130
HASH TABLE	3	.89	3076
ATOMS	16.75	.96	18524
PNAME	4	1.11	5114
INTEGER	1	.89	1025
OTHER	2.5	1	2880
		TOTAL	75100

So the equivalent of a 100 page (51,200 word) working set in Interlisp-10 occupies 75,100 words on the PDP-11. (roughly 66% of the number of bits).

Status

The hardware Memory Mapping Device to map the 22-bit address space of INTERLISP-11 into a 19 bits (512K) real core address has been designed and specified, built, and is now operational.

In terms of software implementation, we have designed the L-code instruction set, implemented it in microcode, and debugged the microcode. This microcode provides the basic primitives for allocating, accessing, and manipulating all the INTERLISP data types as well as the function invocation and stack handling mechanisms. The compiler itself, which is written in LISP, has also been implemented. A substantial portion of the INTERLISP kernel (the part of INTERLISP-10 that is written in machine language) has been re-written in LISP. Another large fraction of the LISP kernel has been written in L-code (rather than in PDP-11 code). We have also completed the work required to modify software that although written in LISP, was machine dependent. These are major steps in the direction of machine independence, and should pay off handsomely when transfer to new, more cost-effective hardware becomes possible.

Consequently, as of the end of the INLAT contract (31 January 1976) there exists an INTERLISP-11 facility that is capable of running LISP code compiled elsewhere, and that provides an interpreter with which users can sit at the PDP-11 console, type in LISP code and execute it, as demonstrated to ARPA-IPTO management on 4 February 1977. However, due to long delays in

hardware delivery and check-out, this implementation fell short of what we expected to have at the end of the contract. Implementation will be completed in a follow-on contract.

SECTION 3 - THE NATURAL LANGUAGE INTERFACE

Introduction

In this Section we describe the Natural Language Interface (NLI) for HERMES commands that was designed and implemented in our INLAT project. The objective of the NLI is to enable users to formulate in English their requests for executing mail manipulation actions, or for explanations on how to perform those actions using HERMES commands. Such a facility would serve two very useful purposes in an Intelligent Terminal environment:

a) an instructional purpose, by showing users specifically how to perform something they have need for, and

b) an executional purpose, by enabling users to circumvent the command language completely and thus make the underlying system accessible to people with little or no experience in its usage.

In order to serve these purposes adequately, the NLI should have the following properties:

1) the NLI should be habitable - i.e. it should be able to deal with requests spanning a wide range of English syntactic constructions within the subject domain, as well as with common errors in spelling or in grammar.

2) the NLI should be efficient - total parsing and interpretation time should be short enough (less than 1 cpu second) to permit comfortable interaction.

3) the NLI should be easily modifiable to deal with any of a number of domains of discourse with a minimum of reprogramming, and the addition of information needed to deal with a new domain should be as simple and straightforward as possible.

A "general solution to the natural language problem for unrestricted discourse" would, a fortiori, satisfy these goals, if we could develop such a system with current knowledge! Unfortunately, that solution is not presently within sight. However, by limiting the domain of discourse it is possible to produce very acceptable and useful NLIs. The SOPHIE system in the domain of electronic troubleshooting, and the LUNAR system in the domain of lunar rock chemical analyses, are cases in point. The system to be described here constitutes another example of an NLI capable of handling pretty sophisticated and "free" English requests, and introduces a novel approach that makes the system more easily extendable to other domains than SOPHIE, and much more efficient to run than LUNAR.

The following two annotated protocols give an idea of the capabilities of the system. The first protocol shows the "explanation" abilities of the system. Users can issue HERMES commands as they would normally, and those commands will get executed and answered by HERMES. But if instead of a HERMES command users type something in English (i.e.: something that HERMES can not digest), the NLI intervenes. Notice that all the requests in the protocol are for explanations; if instead of a

"how to" type question the user had used the imperative form, the request would have been translated into HERMES commands, executed invisibly to the user, and the result would have been shown.

The second protocol gives an idea of the range of English that can be handled by showing how the system deals with a number of common variations on the theme of "messages dealing with a certain topic." Instead of showing the user typing directly to the INLAT, which would have produced monotonous responses, we show the interpretation produced by the NLI.

Hermes/Inlat >>>> Preliminary version 0.0.3 (26-Dec-76 11:21AM)

Please be patient a moment while I start-up
Hermes over the Net and initialize ...

  BBN-TENEX 1.34.20, BBN-SYSTEM-D EXEC 1.54.44
@LOG GRIGNETTI 10099
  JOB 33 ON TTY121 17-Mar-77 16:45
  PREVIOUS LOGIN: 17-Mar-77 16:15
  [JOB 24 ALSO LOGGED IN UNDER GRIGNETTI]

HERMES 2.6.6 13-Oct-76
Type NEWS, followed by carriage return, for the latest news.

OK. Thanks for waiting.

The INLAT establishes a connection over the network, logs the user in, and puts him in contact with HERMES.

****

93\How do I print just the number, the subject, and the author of a message?

The backslash (\) is the INLAT's prompt character, preceded by the interaction number. The user's typing is underlined.

To show the indicated parts of the messages, we create a "Template":
>CREATE TEMPLATE {MESSAGE-NO&SUBJECT&FROM}
>>LINE-INSERT
MESSAGE-NO.+
SUBJECT:+
FROM:+
SEPARATE
<Z>
>>DONE

The INLAT explains how to execute this action by means of HERMES commands. First, it is necessary to create a new template and give it a name (i.e.: {MESSAGE-NO&SUBJECT&FROM}).

Issue the Hermes command:
>PRINT * {MESSAGE-NO&SUBJECT&FROM}

Next, a PRINT command must be issued, to print all messages (*) through the template just created.

****

94\Show me how to print the messages dealing with "Intelligent Terminals" that Steve sent me during July.

Issue the Hermes command:
>PRINT /SUBJECT: "INTELLIGENT TERMINALS"/FROM WALKER/TO: GRIGNETTI/
AFTER 7-1-1976/BEFORE 7-31-1976

Here the PRINT command "filters" the user's messages so that only the ones desired are obtained.

****

95\What do I do to see the last 3 messages that arrived last week?

>SURVEY /AFTER 3-6-1977/BEFORE 3-12-1977
Now, using this ordered list of messages:
PRINT the last 3 elements of the list this survey will produce.

The SURVEY command produces a one-line summary of all the messages received within the indicated period ("today" was March 16th).

****

96\What do I do to see the 3 longest messages that arrived last week?

Since messages are not ordered by size, we must select the ones we want and sort them in order of increasing length.

>CREATE SEQUENCE Some-name#1
>>ADD */AFTER 3-6-1977/BEFORE 3-12-1977
>>SORT LENGTH
>>REVERSE
>>DONE
>SURVEY Some-name#1
Now, using this ordered list of messages:
PRINT the first 3 elements of the list this survey will produce.

The newly created SEQUENCE is SURVEYED.

*** Note that it may or may not be reasonable for you to physically create any filters or sequences illustrated above.

Meaning, if you have just a few messages you can just as well SURVEY them and select the 3 longest by inspection.

****

97\Explain to me how to get a summary of all the messages about "Intelligent Terminals" that I received from Steve, excluding those I've seen already.

Issue the Hermes command:
>SURVEY /SUBJECT: "INTELLIGENT TERMINALS"/FROM WALKER/UNSEEN

UNSEEN happens to be a pre-defined HERMES filter.

****

98\Describe the procedure for printing the first 3 messages sent by either Walker or Brown during the 2nd week of Dec.

>CREATE SEQUENCE Some-name#1
>>ADD */FROM WALKER/AFTER 12-12-1976/BEFORE 12-18-1976
>>ADD */FROM BROWN/AFTER 12-12-1976/BEFORE 12-18-1976
>>SORT
>>DONE

Filters can not be ORed with HERMES. Proper merging is achieved using SEQUENCEs instead.

>SURVEY Some-name#1
Now, using this ordered list of messages:
PRINT the first 3 elements of the list this survey will produce.

*** Note that it may or may not be reasonable for you to physically create any filters or sequences illustrated above.

****

99\Tell me how to get a summary of the messages that arrived between April 15 and the end of June, except for those that are about "Intelligent Terminals."

We will do this one by creating a filter:
>CREATE FILTER Some-name#2
>>REQUIRE DATE: AFTER 4-15-1976
>>REQUIRE DATE: BEFORE 6-30-1976
>>REJECT SUBJECT: INTELLIGENT TERMINALS
>>DONE

Here there is no pre-defined filter so we must create one. (Contrast with 97 above.)

Issue the Hermes command:
>SURVEY */Some-name#2

SURVEYing all messages that pass through the newly created filter.

*** Note that it may or may not be reasonable for you to physically create any filters or sequences illustrated above.

```
----
(ARE THERE ANY MESSAGES ABOUT "INLAT")

[QUESTION                                    (* The input is a QUESTION)
  COMMAND = TRANSCRIBE                       (* The command to be used is TRANSCRIBE)
  MESSAGEITEM =                              (* The arguments of this command constitute
                                                a MESSAGEITEM)
   [NP
     NOUNTYPE = <MESSAGEITEM>                (* See discussion of Case Structure "slots")
     ISEQ = *                                (* The Implied SEQuence is "all messages"(*).
     FILTERS =((SUBJECT: "INLAT"))
     FORMAT = STEMPLATE]                     (* Produce an output formatted as a SURVEY)
  SWITCHES =(EXECUTE YES/NO)]                (* Produce an execution, not an explanation,
                                                and note this is a YES/NO type question)

----
(ARE THERE ANY MESSAGES WITH THE WORD "INLAT" IN THE FIELD THAT
DESCRIBES THEIR CONTENT)

[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ = *
     FILTERS =((SUBJECT: "INLAT"))
     FORMAT = STEMPLATE]
  SWITCHES =(EXECUTE YES/NO)]

----
(WHAT MESSAGES HAVE "INLAT" IN THEIR SUBJECT)

[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ = *
     FILTERS =((SUBJECT: "INLAT"))
     FORMAT = STEMPLATE]
  SWITCHES =(EXECUTE)]

----
(WHICH MESSAGES CONTAIN THE WORD "INLAT" IN THEIR SUBJECT FIELD)

[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
```

```
      NOUNTYPE = <MESSAGEITEM>
      ISEQ = *
      FILTERS =((SUBJECT: "INLAT"))
      FORMAT = STEMPLATE]
  SWITCHES =(EXECUTE)]
```

(GIVE ME THE MESSAGES THAT DEAL WITH "INLAT")

```
[IMPERATIVE
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ = *
     FILTERS =((SUBJECT: "INLAT"))]
  SWITCHES =(EXECUTE)]
```

(I WOULD LIKE TO READ ALL THE CORRESPONDENCE I HAVE ON "INLAT")

```
[CLAUSE
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     FORMAT = PTEMPLATE
     ISEQ = *
     FILTERS =((SUBJECT "INLAT"))]
  SWITCHES =(EXECUTE)]
```

(DO I HAVE ANY MAIL CONCERNING "INLAT")

```
[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ = *
     FILTERS =((SUBJECT: "INLAT"))
     FORMAT = STEMPLATE]
  SWITCHES =(EXECUTE YES/NO)]
```

```
----
(WHO SENT ME MESSAGES DEALING WITH "INLAT")

[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ = *
     FILTERS =((SUBJECT: "INLAT")
               (TO: GRIGNETTI))
     FORMAT =(FROM)]
  SWITCHES =(EXECUTE)]



----
(WHEN DID I GET MY FIRST MESSAGE ON "INLAT")

[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ =(SEQUENCE (SUBSET 1:1))
     FILTERS =((OR (TO: GRIGNETTI)
                   (CC: GRIGNETTI)))
               (SUBJECT:  "INLAT")
     FORMAT =(DATE:)]
  SWITCHES =(EXECUTE)]



----
(PRINT ONLY THE FIELD THAT DESCRIBES THE CONTENT OF THE MESSAGES I
RECEIVED YESTERDAY)

[IMPERATIVE
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     FORMAT =(SUBJECT:)
     ISEQ = *
     FILTERS =((YESTERDAY))]
  SWITCHES =(EXECUTE)]

----

(WHAT MESSAGES DID I RECEIVE YESTERDAY CONCERNING "INLAT")
```

```
[QUESTION
  COMMAND = TRANSCRIBE
  MESSAGEITEM =
   [NP
     NOUNTYPE = <MESSAGEITEM>
     ISEQ = *
     FILTERS =((SUBJECT: "INLAT")
               (YESTERDAY))
     FORMAT = STEMPLATE]
  SWITCHES =(EXECUTE)]
```

Background

Like all natural language systems, our NLI makes use of syntactic, semantic, and pragmatic knowledge. As a first step in describing the interaction of these types of knowledge in the INLAT NLI, we will first consider some characterizations of the use of these three types of knowledge in natural language systems in general. One very common view, in first approximation, is the following: Syntactic knowledge is used to assign to the linear string of words which is the input to the NLI, a structure which groups the words according to the hierarchical syntactic relations among them. The purpose of semantics, then, is to assign a "meaning" to such a syntactic structure. A well-designed syntactic component can greatly simplify the design of this semantic component, since it can transform syntactic variants of a phrase into a common form. The role of pragmatic knowledge is to assign a "purpose" to the utterance (in some context) and perhaps to determine an appropriate response given the assigned purpose. This view can be schematically represented as a simple pipeline, in which text comes in at one end, is processed by a syntactic component, the resulting syntactic structure is processed by a semantic interpreter, and the semantic interpretation is used in turn as the input to a pragmatic processor. This is the way LUNAR works.

A valuable feature of this model, related to goal 3), is the fact that the process of applying semantic knowledge is entirely independent of the process of using syntactic knowledge. The

interface between the two knowledge sources is entirely defined by the permissible structures produced by the syntactic process. Once this has been defined, syntactic and semantic knowledge and processing structures can be built independently, and since there is good reason to believe that the syntactic regularities of English extend over many subject domains, changing the subject domain of the NLI does not require much modification of the syntactic component.

The view outlined above stresses the interpretive or descriptive aspects of the use of semantic and pragmatic information in an NLI - semantics is used simply to provide an interpretation of the structure produced by the syntactic component. Recent work on practical NLI's (such as the "semantic grammar" used in the SOPHIE system) has emphasized another aspect of the interaction of syntactic processing with semantic and pragmatic knowledge. This aspect can be characterized as the "proscriptive" use of semantics and pragmatics. The meaning of a syntactic structure is, in general, built up by combining the meanings of sub-structures in some fashion. Not all syntactically well-formed structures can be interpreted (particularly within a restricted subject domain), and thus one can conceive of semantic knowledge which characterizes the collections of "syntactic modifiers" which can be meaningfully applied to a given syntactic head. Similarly, at any point in a dialogue with the user, there are semantically interpretable structures that are pragmatically impossible or implausible. This information is of potentially great use for controlling syntactic

processing - both in guiding the search for syntactic structures, and in restricting the range of structures produced by the syntactic process. Such a control necessitates a much greater level of interaction between the (knowledge embedded in the) semantic processor and the details of the syntactic process - and unlike the "pipeline" approach in which only the characteristics of the final output of the syntactic process need to be considered in the design of the semantic processor, in this approach some portion of the set of control decisions made within the syntactic processing must be documented and externally accessible.

One way to do this is to completely merge the three process and knowledge structures. The "semantic grammar" used in SOPHIE, and the "pragmatic grammar" used in HWIM, are examples of this approach. To characterize such grammars, note that the basic operation in most linguistic systems can be viewed as a prediction and/or search for a linguistic unit suggested by contextual information (such as the structure of units already found, the state of the syntactic processor, the state of the overall dialogue, etc.). In a syntactic processor the linguistic units are such syntactic structures as NP's, PP's, CLAUSE's, PREP's, NOUN's, etc. In "semantic grammars," the linguistic units are generally phrases with a constrained meaning, e.g. a phrase referring to a circuit component or one referring to an electrical measurement in a circuit. Such units often (but not always) correspond to a single syntactic unit. As these units are discovered they are incorporated into more complex semantic units

in the same way PREP's and NP's are incorporated into PP's syntactically. Note that in this approach both the control (proscriptive) and interpretive (descriptive) aspects of semantics and pragmatics are tightly enmeshed in the search for units, and there is no explicit general syntactic processing evident. In fact, it is often difficult in such systems to take into account regularities in the permissible variations of a given unit (e.g. the equivalence of passive and active constructions in all semantic units represented by clauses with passivizable verbs).

An alternative approach would be to maintain the syntactic orientation of the basic prediction and search process, and separate the proscriptive aspects of semantic and pragmatic knowledge from the interpretative aspects, having the syntactic processing interact only with the proscriptive semantics and pragmatics. This leaves the interpretive semantics potentially as independent of processing as the pipeline approach. Many NLI's based on "case structure grammars" can be viewed in this light.

The difficulty with such an approach lies in part in the problem of writing case structure rules that capture anywhere near as much of the known domain constraints as the "semantic grammar" systems, since without this it is difficult to obtain systems with wide grammatical flexibility that do not suffer from combinatorial efficiency problems produced by syntactic ambiguity (particularly the ambiguity in placement of modifiers).

The INLAT NLI Approach

The system uses an Augmented Transition Network (ATN) grammar together with a case-oriented dictionary to achieve a close and efficient integration of the syntactic processing with the case structures (which include semantic and pragmatic properties of objects).

The ATN defines, using normal syntactic categories, a very general surface structure of about the capability of the LUNAR and GSP systems. If case structures and semantic information (including interpretation rules) are omitted from the dictionary, the grammar functions as a standard parser, producing closely related "deep structures" for syntactic paraphrases.

The system provides mechanisms for users to define semantic interpretation rules and case frame checks which are to be applied at various points in the parsing process. Thus constituents are interpreted as soon as they are parsed, and the structure of the semantic interpretations thus produced are checked when filling the case frames for higher structures. Since the "most likely" local interpretation may not fit the case requirements of containing structures, the system provides a general coercion mechanism to reinterpret a constituent in light of its context when necessary. To facilitate reinterpretation, as well as certain anaphoric references, the original syntactic structure is maintained throughout the parsing together with any semantic interpretations. This organization permits the semantic and pragmatic interpreter to tightly control the syntactic parsing at

all levels, from the lowest phases up through the association of modifiers and complex nominal and verb structures. It reduces ambiguity at all levels, thus improving the overall efficiency of the system. It also guarantees that the first parse that comes out is interpretable within the discourse domain (and is actually interpreted as it comes out). At the same time, the syntactic processor and the semantics and pragmatics are cleanly and completely separated. The semantics and pragmatics for a given domain can be coded without detailed knowledge of the operation of the syntactic processor, and yet still maintain the tight control that we desire to limit ambiguity. This fact facilitates building systems for new domains by modifying part or all of these semantics and pragmatics and maintaining the syntactic processing constant.

The syntactic processor greatly extends the range of the syntactic structures which we can handle to include such structures as complex relative clauses, complements, comparatives, passive transformations, extraposition, ellipsis, and gapping.

How the System Works

Having reviewed the structure of the parsing system, we present next a description of the parsing process and of the way this process is controlled.

The parsing process is based on the notion of incremental interpretations, i.e.: building a phrase by adding constituents one by one and interpreting the results. As each new potential

syntactic constituent is found, semantic and pragmatic knowledge is used to check if it is consistent with the interpretation of the phase already built. If so, the semantic interpretation of the constituent is incorporated into the semantic interpretation of the phrase.

The semantic and pragmatic knowledge resides in a data-structure associated with the head word of a phrase - the main verb of a CLAUSE or the head noun of a Noun Phrase (NP). This data-structure can be viewed as defining a set of "slots." These "slots" can be filled by modifier constituents such as Prepositional Phrases (PPs), NPs, relative clauses, etc. If these constituents satisfy a set of semantic and pragmatic criteria associated with the "slot," then the meaning of the modifier constituent can be incorporated into the semantic interpretation of the main phrase. This notion of restricting the type of phrase which may be placed in modifier slots is closely related to the notion of a Case Frame as given by Fillmore and others (see Bruce [76]). The semantic and pragmatic checks associated with a given slot are greatly facilitated by the fact that the lower level phrases are completely interpreted before they are presented to the Case Frame Checker. This means that the criteria can be based on the interpretation of phrases rather than on the syntactic form of these phrases. In particular, NPs are interpreted as they are built and they are assigned to one or more categories referred to as NOUNTYPES. Most of the Case Frame Checks within our system are in fact single tests of the NOUNTYPES of NPs or on the NOUNTYPES of prepositional objects in prepositional phrases.

The following figure shows some examples of Case frame "slots:"

```
SEND

        (VERBCASES
            [[FROM  ((= (POBJ NOUNTYPE)
                        <USER>)
                     (FILTERS _ (BUILDQ (FROM: #)
                                        (GetPATH * POBJ HEAD HEAD]
             [SUBJECT ((= (NOUNTYPE)
                          <USER>)
                       (SUBJ <= *)
                       (FILTERS _ (BUILDQ (FROM: #)
                                          (GetPATH * HEAD HEAD
             [INDOBJ ((= (NOUNTYPE)
                         <USER>)
                      (INDOBJ <= *)
                      (FILTERS _ (BUILDQ (TO: #)
                                         (GetPATH * HEAD HEAD]
             (OBJECT ((= (NOUNTYPE)
                         <MESSAGESEQ>)
                      (OBJECT <= *)))
             (TIMEMOD (T (TIMEMODS <- (GETTIMEMODS *]
        QUESTIONINTERP QINTERP-MESSAGEVERB
        TIMEVERB T))



MESSAGE

          (POSTMODCASES
              [[WITH ((IN (POBJ NOUNTYPE)
                          (<FIELDNAME> <IDENTIFIER> <STRING>))
                      (FILTERS _ (MAKEFILTERFROMPP *]
               [(ON ABOUT)
                ((IN (POBJ NOUNTYPE)
                     (<SUBJECT> <STRING>))
                 (FILTERS _ (BUILDQ (SUBJECT: #)
                                    (GetPATH * POBJ HEAD HEAD]
               [FOR ((= (POBJ NOUNTYPE)
                        <USER>)
                     (FILTERS _ (BUILDQ (TO #)
                                        (GetPATH * POBJ HEAD HEAD]
               [OF ((= (POBJ NOUNTYPE)
                       <USER>)
                    (FILTERS _ (BUILDQ (OF #)
                                       (GetPATH * POBJ HEAD HEAD]
               [(FROM BY)
                ((= (POBJ NOUNTYPE)
                    <USER>)
                 (FILTERS _ (BUILDQ (FROM: #)
                                    (GetPATH * POBJ HEAD HEAD]
               (VERBMOD (T (FILTERS <- (GetPATH * FILTERS]
          (PREMODCASES
```

```
     [(ADJ ((INEVAL (HEAD)
                    FILTERADJS)
            (FILTERS <- (InterpADJasFILTER *]
 NOUNTYPE
    (<MESSAGESEQ>)
IDENTIFIABLE T
NUMERABLE T))
```

We can view the parsing of a phrase (e.g. an NP or CLAUSE) as a process of searching for syntactically well-formed substructures (either words or phrases) and assigning them to case structure "slots" associated with the phrase. As an example, consider the clause:

John gave the red-headed boy the book last Tuesday.

As a simplification we can consider that the verb "gave" fills the HEAD slot of this clause, the NP "John" fills the SUBJECT slot, the NP "the red-headed boy" fills the INDIRECT OBJECT slot, the NP "the book" fills the OBJECT slot, and the NP "last Tuesday" fills a TIME-MODIFIER slot. At any time in the parsing of a phrase there are a limited set of syntactic structures that are plausible to look for. This set depends on such things as:

a) general restrictions on all phrases of the given class, based on the set (but not the contents) of currently assigned slots

b) the next word in the string

c) the actual contents of the currently assigned slots.

Thus, if we are parsing a top-level clause and have assigned no slots, it is plausible to look either for an NP (which may eventually fill the SUBJECT or OBJECT slot), or an untensed verb (which will fill the HEAD slot and mark the clause as IMPERATIVE), or a prepositional phrase which may fill any of several modifier slots, etc.

In the current version of the parser, NPs are handled in the following fashion. From the point of view of the syntax, a NP consists of the following parts: a determiner (DET), a sequence of pre-modifiers (PREMODS), a HEAD, and a sequence of post-modifiers (POSTMODS). In the noun phrase "the three largest polished pewter candlesticks from California that I have seen," the DET is "the three largest" (containing the article, the number, the superlative, and potentially an ordinal), the PREMODS are "polished" and "pewter" (and in general include adjectives, associated adverbs, and present participles as well as past participles and nouns), the HEAD is "candlesticks," the POSTMODS are "from California" and "that I have seen" (and in general include prepositional phrases and relative clause modifiers). Any of these parts may be missing (though the current system will not allow POSTMODS if there is no HEAD, and will thus not handle NPs like "the largest I have seen"). Each part is defined both by its internal syntactic structure (e.g. prepositional phrase or adverbially modified adjective) and by its relative position in the NP. The syntactic processor searches for the legal syntactic structures in each position (e.g. it first looks for a DET, then looks for PREMODS followed by a noun for HEAD, followed by PPs and relative clauses). As each of these parts is found, it is passed to the semantic case assignment processor. It is the responsibility of the case checker to see if the syntactic structure passed to it can plausibly fill the part specified by the syntactic processor, given both the set of already filled parts and semantic and pragmatic knowledge associated with the HEAD.

Let us consider the processing of the sentence:

"Print the messages from Woods from June 1 to the end of last month on the LPT:"

The syntactic processor notes that the first word in the sentence is an untensed verb, so it assigns that verb as the HEAD of the sentence and sets up to parse an imperative sentence. Since the dictionary entry for "print" indicates that it is a transitive verb, the parser next searches for an NP to be the OBJECT. In fairly direct fashion the article "the" is assigned as the determiner for the NP, and the noun "messages" is assigned as the HEAD (since there are no numbers, ordinals or superlatives, the DET is quite simple, and there is no trouble determining the HEAD since there is only one noun following the DET). The processing becomes more interesting once the parser attempts to find the POSTMODS for "messages." By checking the case frame for "message" it determines that PPs with preposition "from" can act as post-modifiers for "message," so it begins to search for such a PP. In searching for the NP following the preposition "from" we see the first use of case frame information to restrict search - once the HEAD "WOods" is found, it is noted that this word does not allow any post-modifiers (in general, the use of a post-modifier after a proper noun is unlikely, and in the mail-system context it is pragmatically impossible). Thus, even though there is a well-formed PP immediately after "Woods," the parser will not even attempt to look for it (or any other post-modifier). Since there are no more constituents to search

for, the NP "Woods" is interpreted by the semantic processor, giving a representation containing the NOUNTYPE <USER>, and with the name "Woods" as the HEAD (this interpretation is produced by the interpretation function associated with the case frame for the HEAD "Woods"). Once the NP has been completely processed, the parser reverts to the PP level and consults the case frame for the preposition "from." Since the result of interpreting the NP is not a <DATE> or <TIME> expression, the case frame for "from" indicates that the PP is completed, and the result is a structure of the form

[PP PREP FROM POBJ [NP NOUNTYPE <USER> HEAD WOODS]]

(as well as some added structure which is irrelevant to the current example). Since the PP was found as a result of a PUSH arc in the NP network which is looking for a POSTMOD for the HEAD "message", this PP is checked to see if it satisfies the CASEFRAME for "message". This is done by passing to the CASEFRAME processor the PP, the CASEKEY FROM (the CASEKEY is an indication of the syntactic/semantic relation between the HEAD and a new phrase - it may be a preposition or a relation like TIMEMOD, SUBJECT or INDIRECT OBJECT), along with the CASEFRAME for the HEAD. The case frame for "message" indicates that a PP whose preposition is FROM is acceptable if the NOUNTYPE of the prepositional object (POBJ) is <USER>, so the PP "from Woods" is acceptable. The case frame indicates that the result of adding such a postmodifier to "message" should be to add to the case slot FILTER the filter specification (FROM *user*) where *user* is the HEAD of the POBJ,

in this case Woods. At this point the grammar indicates that another PP may occur as a POSTMOD of "message", so the parser searches for another PP. The parser finds the preposition "from" and then searches for an NP as the POBJ. When no article or adjective is found, and the first word is "June", whose dictionary entry indicates that it is a month, the parser enters the special date section of the NP network which parses "June 14" as a special type of NP with NOUNTYPE <DATE>. While the grammar allows dates to be given without an explicit year, in the HERMES context it is necessary to know the year. Thus, the caseframe associated with the NOUNTYPE <DATE> has an interpretation function that checks to see if an explicit year has been given (e. g. as in "June 14, 1976") and if not it computes the most plausible year in the HERMES context (we assume that all dates refer to the past, and are as recent as possible, so the year is either the current year or the previous year, depending on whether the date has occurred during the current year). The net result of this process is to produce an NP of the form

[NP NOUNTYPE <DATE> HEAD JUNE DAY 4 YEAR 1976]

which is returned as the POBJ of the preposition "from". The grammar indicates that if a PP is found with the PREP "from" and the NOUNTYPE of the POBJ is <DATE>, then it is possible for the PP to have a PP complement of the form "to <DATE>", so the parser uses the PP network to search for such a PP. Once the PREP "to" is found, the parser uses the NP network to search for an NP which can be interpreted as a <DATE>.

In traversing the NP network, the article "the" is found, followed by the noun "end" which is taken as a possible HEAD for the NP. The caseframe for "end" indicates that it can take a PP as a POSTMOD, so the parser uses the PP network to search for a PP. The PREP "of" is found, and then the NP network is used to search for the POBJ. Since there is no article to start the NP, the NP network guides the parser to save the adjective "last" as a PREMOD (NP premodifier), and then to find the noun "month" which is taken as a possible HEAD for the NP (since there is no noun following it). The CASEFRAME for "month" indicates that there are no acceptable POSTMODS in the HERMES context, so the parser completes the parsing of the NP by applying the interpretation function which is given in the CASEFRAME for "month" to the preliminary NP, yielding:

[NP NOUNTYPE MONTH PREMODS ([ADJ HEAD LAST]) HEAD MONTH].

The interpretation function checks through the PREMODS and when it sees the adjective "last" it computes a representation for the NP as a specific month, which depends upon when the phrase "last month" is typed. If, for example, we assume that the phrase is typed in June 1976, then the result of the interpretation function is the NP

[NP NOUNTYPE <MONTHNAME> HEAD MAY YEAR 1976].

This NP is returned as the POBJ of the PREP "of", and the resulting PP

[PP PREP OF POBJ [NP NOUNTYPE <MONTHNAME> HEAD MAY YEAR 1976]]

is passed up as a possible POSTMOD for the HEAD "end". The CASEFRAME for "end" accepts a PP with PREP "of" and whose POBJ has NOUNTYPE <MONTHNAME>, and it indicates that the HEAD of the current NP should be changed from "end" to the HEAD of the POBJ (in this case MAY), that the slot YEAR should be copied into the current NP, and that the slot DAY should be filled with the last day of the given month (31 in the case of MAY). The net result of all this is that the case structure for the phrase "the end of last month" becomes

[NP NOUNTYPE <DATE> HEAD MAY DAY 31 YEAR 1976].

Since this head cannot take any POSTMODS in the HERMES context, and after the interpretation function associated with <DATE> checks and finds there is an explicit YEAR, the case structure is passed up as the POBJ of the preposition "to". The grammar indicates that such a PP is complete, and the resulting structure

[PP PREP TO POBJ [NP NOUNTYPE <DATE> HEAD MAY DAY 31 YEAR 1976]]

is popped and taken as the PP complement of the prepostion "from" whose POBJ was seen previously to be "June 14". The grammar notes that this is a special PP whose POBJ is a <DATE> and produces the interpretation

[TIMEMOD RELATIONS

([PP PREP AFTER

[NP NOUNTYPE <DATE> HEAD MAY DAY 1 YEAR 1976]]

[PP PREP BEFORE

[NP NOUNTYPE <DATE> HEAD JUNE DAY 14 YEAR 1976]])]

as the result of the push to the PP network from the NP with head "message". The CASEFRAME for the HEAD "message" indicates that it can accept a POSTMOD with CASEKEY TIMEMOD, and causes the filter ((AFTER MAY-1-1976)(BEFORE JUN-14-1976)) to be added to the FILTERS slot.

The processing continues with another round of POSTMOD chasing, but the PP

[PP PREP ON POBJ [NP NOUNTYPE <DESTINATION> HEAD LPT:]]

ends up having a CASEKEY which is not acceptable in the HERMES context as a modifier of "message". However, popping up to the main verb level, the CASEFRAME for "print" indicates that the PP can be accepted as a filler of the slot DESTINATION. This completes the parsing.

SECTION 4 - THE INLAT MONITOR

Introduction

In what follows, it is assumed that the reader is familiar with the general goals and philosophy of the Inlat for the Intelligent Terminal. It is the purpose of this section to detail progress made on the Inlat Monitor.

The job of the monitor, within the Inlat framework, is to set up all necessary interfacing (over ARPA network) that will allow the user, talking to an Interlisp job, to be able to interact both directly and indirectly with the BBN-Hermes message system*.(1) The monitor must therefore establish the necessary network connections that enable the Interlisp-to-Hermes communication. It also must read all characters being transmitted over the network, in both directions, and determine what is to be done with those characters. In particular, the Inlat's primary purpose is to provide tutoring and assistance in the use of Hermes--indicating that many user inputs will be addressed to the Inlat itself (e.g. questions about how to use some feature of Hermes), and will henceforth be referred to as 'Inlat commands.' But the user must also be able to engage in a direct dialogue with Hermes. Thus, on the user side of the network, the monitor must essentially be able to distinguish three different types of input: 1) an Inlat command, to be passed to the Inlat parser, etc. for processing;

(1) *The Inlat is being designed for use with many diverse systems, but currently our prototype is concentrating only on Hermes.

2) a Hermes command to be passed directly to Hermes to act upon; and 3) the middle ground, viz. an intended Hermes command that is syntactically or semantically invalid and cannot be acted on by Hermes.

On the other side of the Network are the Hermes (and Tenex) responses. These include such things as echoes and file confirmation requests, which must be gobbled up by the monitor; a response to a user initiative which must be displayed on the user's terminal; responses to Inlat initiatives, which must be returned to the requesting module but not be displayed on the console; questions that Hermes wishes to ask of the user (e.g. DELETE MESSAGES AFTER FILING?) which must be displayed and the user then put in touch with Hermes to give the appropriate answer; and so on and so forth.

The discussion above describes the essential mode of operation that the monitor must establish and the type of recognition abilities that it must possess for correct handling of the various pieces of dialogue that are expected to occur during an Inlat session. This is accomplished primarily by the monitor containing a full Hermes parser and simulator which enables it to decide whether or not a user input is intended as a Hermes or an Inlat command, and if it is a Hermes command, whether or not it will be accepted by Hermes. It should be noted that wherever possible, the monitor will actually correct an incorrect Hermes command (see Error Correction). The Hermes simulator allows the monitor to catch (and often correct) semantic errors, since at all times it has a perfect model of the Hermes environment.

The monitor also is keeping detailed history lists of each significant interaction the user initiates, thereby enabling a very useful and dynamic sort of assistance in regard to mailbox handling (as well as maintaining a good profile for enhancing the appropriateness of Inlat responses with respect to questions about Hermes). In the following subsections we will describe the various capabilities of the monitor, indicating the features that ultimately will be made available in the Intelligent Terminal.

Hermes Simulation

The mode of operation of the Inlat is that the user types in commands which are delegated either to Hermes or the NLF parser. In the event that the user is intending to address the command directly to Hermes, the Inlat monitor that is reading the input and making the decision about what to do with it, must respond to the user exactly as Hermes would, thus giving the appearance of direct communication with Hermes. In order to mimic Hermes responses, the Inlat must possess a complete parser for the Hermes command language. Obviously, this parser is also necessary for making the decision as to whether or not the input should be passed to Hermes. There are several reasons why the monitor must be able to parse Hermes commands. First, as stated above, the monitor must be able to recognize Hermes commands, and to mimic Hermes responses (word completions and prompts). An alternative mode of operation, viz. placing the user in direct communication with Hermes via the ARPAnet, was tried and proved to be infeasible. This was because of the very lengthy transmission delays making echoing (and recognition/prompting) absurdly slow.

We have to use the ARPAnet for Interlisp to Hermes communication because of the design constraint that the Inlat must run on the PDP-11 and Hermes on Tenex.

Because the Inlat must be able to aid and advise the user in regard to difficulties he might be having in the use of Hermes, it is essential that the Inlat know the current state of Hermes and the mailbox, as well as how that state was arrived at, i.e. the history. Clearly this can only be done if the Inlat can monitor the session in the sense of overseeing the various transactions and interchanges that take place. This means being able to parse the utterances issued in both directions of the dialogue.

Finally, since one of the primary objectives of the Inlat is to provide an instructional facility, the ability to understand the commands that a student has issued is crucial in being able to evaluate a student's performance on some task, both for simple scoring and critiquing.

With these goals in mind, the Inlat monitor has incorporated into it a full parser for the Hermes command language. In as much as many Hermes commands and command fields

are passed to Tenex to read (e.g., file names, utility commands such as DIRECTORY, JOBSTAT, DAYTIME, etc.), this means also simulating certain portions of the Tenex EXEC. (One advantage of this Tenex simulation is the ability to do spelling correction on file names.)

Aside from the fundamental necessity of the simulation for the required mode of operation of the Inlat, the Hermes simulation is essential for one of our primary research goals, viz.,

modeling. There are three models that must be constructed and maintained in order for the Inlat to exhibit 'intelligent' behavior in the context of the scenario for which it is being designed. These models are: 1) Hermes; 2) the user; and 3) the environment.

By being able to parse all Hermes commands, the system is able to know exactly what action Hermes is going to take at all times. This simulation constitutes, in fact, the Hermes model. More importantly, it provides also for the third model, the environment. Being able to model Hermes implies being able to model the environment, since at all times we know its state and how it is being acted upon.

Hermes maintains (several) user profiles containing idiosyncratic information about each user's mode of operation, etc. The Inlat also maintains a user profile (which will be the 'user model'). Eventually we intend that the Hermes profiles will be contained within the Inlat profile. Presently, those parts of the Hermes profile that are essential to the operation of the Inlat (certain pre-defined templates and filters, switch settings, etc.) are obtained from Hermes during initialization. As the monitor is parsing a Hermes command, it performs a 'semantic pass' that updates various state variables in its model of the environment. This, of course, is how so-called 'semantic errors,' alluded to earlier, are detected. Eventually, this state model will prove most useful in answering user questions about why something 'strange and unexpected' occurred, or in student evaluation and critiquing.

Error Correction and HERMES Extensions

Perhaps the most fundamental precept of the Inlat is the notion of a co-operative, friendly system, free of the rigidity and unforgiving nature that has traditionally characterized computer systems. At one extreme, the Inlat allows the user to phrase Hermes commands very freely. E.g.

"SHOW ME THE LAST THREE MESSAGES THAT WALKER SENT ME YESTERDAY"

which the Inlat NLF parser will translate into the appropriate Hermes commands for the user. However, it is often the case that the user is trying to type in a valid Hermes command--one that should require no translation--but, being human, he makes an error.

In the spirit of DWIM*,(1) the Inlat monitor, while performing its simulation of the Hermes parse, makes certain efforts to see that minor syntactic errors are corrected automatically. In Hermes, this often proves to be quite simple, since it is so rare that the input is ambiguous. Since the Hermes syntax is so simple and straight forward, about the only real syntactic confusion for the user that is found is specifying 'sequences' or Hermes message lists. The goal throughout has been to make Hermes a more 'friendly' system, i.e. one that is forgiving and instead of just telling you that you did something wrong, does its best to make sense out of what was said.

(1) * DWIM (&Do &What &I &Mean), Teitelman, 1969. This is the crucial error correction package in Interlisp.

There is within the Inlat a natural language front-end (NLF). In the event that an intended Hermes command contains an error that for any reason the monitor is unable to fix, because of ambiguity or whatever, the NLF will be able to make an attempt at understanding the command, or perhaps questioning the user to reconcile difficulties. Thus, the error-correction burden of the monitor is clearly limited: those things which can be easily handled are corrected rather than calling a high-powered (and sluggish) general parser. Thus, the monitor does what it can do easily and is not concerned that other seemingly simple corrections go uncorrected. They are, of course, detected.

In accordance with the objectives as described earlier, we reiterate that as the monitor is reading a user input, it is simultaneously trying to recognize it as: a) a correct Hermes command; b) an intended Hermes command which is syntactically (or semantically) invalid; c) an English-like utterance (Inlat command). Furthermore, since in the event that the input is of type (a) or (b) the monitor must be able to mimic Hermes (and Tenex) with respect to recognition and completion of various words and prompts, the input must be processed character by character and decisions made long before the input is complete. There is no possibility of look-ahead!

Correcting errors in message sequences:

The most visible assistance offered by the monitor is found in typing Hermes sequences, or message-lists (messages to be printed, filed, deleted, surveyed, etc.). While the syntax of

sequences is relatively simple, many people find it unnatural to use and very difficult to communicate (teach). In the Inlat, most of the problems with sequence specifications have been done away with.

While sequences as such in Hermes, are already fairly powerful devices, they have been made even more powerful in the Inlat by the addition of several new concepts, and great simplification of use. To begin with, there is almost no syntax. Commas, colons, semi-colons, slashes, blanks, are almost interchangeable (except, of course, where the user wishes to override the Inlat default). For example one can type:

```
PRINT 1 3 LAST FILTER 3-LAST AFTER FRIDAY FROM JIM 1-10 UNSEEN:
      | |   |     | |   |       |   |       | | |     |
```

which would be translated into:

```
PRINT 1,3,%/FILTER;3:%/AFTER 21-MAY-76/FROM JIM;1:10/UNSEEN
```

The vertical bars indicate places where Hermes would fail. Note that the Inlat command has, aside from blanks, only two break characters (the dashes) and even those are somewhat arbitrary. The Hermes line, on the other hand, has 10 mandatory punctuation characters besides blanks. We should note here that one is, of course, free to use as much punctuation as one wishes, and that, above all, the Inlat must accept anything that Hermes will.

Implied Sequences:

An important new feature is what we call the 'implied sequence.' It is often the case that one wants to specify "the last 3 messages from X" or the "first message from Y" or the "first message on Tuesday", etc. While this sort of thing is surely possible to do in Hermes, it cannot be done directly with a single command, but requires creating a sequence, perhaps sorting it, listing (SURVEYing) it, and manually reading off the desired message numbers. The Inlat provides the capability of referring to this 'implied' sequence rather than having to create it and examine the survey of it. While this, too, is essentially syntax-free, we have introduced the back-slash (_\) as a new punctuation character that specifies the creation of an 'implied sequence.' For example, to refer to the first message on Monday, one can simply state

a) 1 _\ on Monday (or 1 on Monday)

The last 3 messages from Walker can be specified:*

(1)

b) %-2:last _\ from Walker

(or other variations if one doesn't like the per-cent sign, etc.) Soon one will be able to say things such as:

LAST 3 (messages) FROM JONES

One might have noticed that in example (b) we have used '-' as an arithmetic operator. One can use '+' or '-' in sequences as operators. Thus one can say:

c) PRINT .:.+5

PRINT LAST - 4

and so on. It is intended that soon we will be able to allow the user to give names to messages (and anything else that one might want to name) and these names, where they evaluate to numbers, can be used in arithmetic expressions.

The careful reader might have noticed that we are already using '-' in place of the colon for 'range' specifications. A little thought, however, will show that it is not ambiguous which use of '-' is intended (especially if an expression has only 1 operator in it). There are however potential ambiguities with the use of these operators for multiple purposes (see further use of '+' below). For example, (b) could also have been written as

---

(1)
*To clarify the distinction between slash and back-slash, note that in Hermes "%-2-%/FROM: WALKER" (assuming Hermes allowed arithmetic operators) would mean: "from among the last 3 messages in the mailbox, select those that are from Walker," while the Inlat phrase "%-2-%_\FROM: WALKER" means: "from among all the messages from Walker, select the last 3."

"%-2-%_\Walker", or (c) written as "PRINT#.-.+5". Here we must restate our objective. In the spirit of DWIM, the Inlat makes its best guess. Lacking sufficient unambiguous syntax, it strives to make that guess 'reasonable'; we certainly can never guarantee that it will be right. If that guess seems obvious, it will just go on; if it is at all questionable it will type out its interpretation in parentheses; if it seems genuinely ambiguous will inform the user and suggest his restating it in English so that it might be handled by the NLF. The parser and corrector never ask questions, though. When there is possible confusion, the interpretation is typed so that the user can back over it (using _Q, _W, _A, or <DEL>) if it is not what was intended.

Filter disjunction

Another new feature is the ability to OR filters together. In Hermes, when filters are strung together, e.g.

*/FILT1/FILT2/ ... /FILTn

the interpretation is the conjunction of the n filters. There is no direct way of forming the disjunction of several filters. The Inlat provides this feature with the '+' operator. To specify the disjunction of several filters, one types:

.../FILT1 + FILT2 + ... + FILTn

(the blanks, of course, are optional). In the event that the back slash is used, then the resulting disjunction of filters is sorted by message number, so that e.g.

SURVEY 1-10_\FROM WALKER + FROM GEORGE

will take the first 10 messages of the merged sequences:

*/FROM WALKER and */FROM GEORGE

Besides greatly alleviating the syntax of sequences, the way that one specifies a date has been considerably improved. It is, again, very much free format and noise words may be generously interspersed. Below are a series of examples of date specifications and their interpretations. Since many of these dates are relative, we will assume that 'today' is Tuesday, 25 May 1976*.

(1)

User Input (2)	Inlat Interpretation**
on last Saturday	*/ON 22-MAY-76
since Thurs	*/AFTER 20-MAY-76
in Jan	*/AFTER 31-DEC-75/BEFORE 1-FEB-76
during Mar	*/AFTER 29-FEB-76/BEFORE 1-APR-76
since the 3rd	*/AFTER 3-MAY-76
since the 28th	*/AFTER 28-APR-76
during July of 75	*/AFTER 30-JUN-75/BEFORE 1-AUG-75
on Friday the 13th	*/ON 13-FEB-76
on Sat, May 13th	*/ON 15-MAY-76 (lets user know that 13th fell on Thurs but day of week overrides date)
before 5-15	*/BEFORE 15-MAY-76
before 15-5	*/BEFORE 15-MAY-76
on Feb 30, 1976	*/ON 29-FEB-76 (lets user know that there were only 29 days in month)
on May 15th, 1976	*/ON 15-MAY-76
since Oct	*/AFTER 31-OCT-75
before Mar	*/BEFORE 1-MAR-76
last week	*/AFTER 15-MAY-76/BEFORE 23-MAY-76

(1) * Hermes, in the latest release, now also defaults the "*/" when only a filter appears.
(2) ** Actually, Hermes defines AFTER to mean "ON or AFTER", so that all of the AFTER dates shown here would really be off by one day.

Displaying

One other extension to the Hermes command language is worthy of mention here. We have added a command called DISPLAY, which can be used to examine specific header fields of messages conveniently. In Hermes one often wishes to look at one specific field of a message. This can be done, of course, but it requires creating a special template. For example, to examine the 'classification' field of message 34, one would have to issue the following Hermes commands:

```
>CREATE TEMPLATE CLASS
>>LINE-INSERT
CLASSIFICATION+
_Z
>>DONE
>PRINT 34 CLASS
```

The DISPLAY command makes this template unnecessary. The syntax of the command is

DISPLAY <message> <fields>

where message defaults to CMESSAGE, and field defaults to SUBJECT:. Several fields may be specified by seperating them by blanks and/or commas. Thus to see the CLASSIFICATION of message 34, in the Inlat one would type:

DISPLAY (messages) 34 (fields) CLASSIFICATION

Archive and History Facilities

Since it is intended that the Inlat will be available to each individual user over an extended period of time, we would very much like to take full advantage of the clerical abilities of the computer and have the Inlat utilize them in a highly personal manner. We are endeavoring to accomplish this personalization by capitalizing on the perfect memory available to us. The primary tools for this personalization are the history lists that record all user interactions, and the archive file which the user can view (if indeed he/she is even aware of its existence) as an infinite waste basket.

The purpose of the history list is twofold: 1) it allows the Inlat to be of dynamic assistance in the handling of the mailbox, and 2) it garners personal information about the user, such as his/her background, level of sophistication, types of errors commonly committed, idiosyncracies, etc. This second purpose, while extremely important to us, must currently be deferred. This aspect will require a good deal of thought and development of heuristics appropriate to abstracting "between the line" information contained in dialogues.

Dynamic assistance is accomplished chiefly in two obvious ways:

a) independence from deletion (all messages that were ever sent or received are archived), and

b) by remembering actions performed with received messages, and messages sent out inadvertently without keeping a copy for the sender. This enables, for example, the handling of requests such as

> Where did I file the message from Craig about "Intelligent Terminals"?
>
> Did Craig receive a copy of the message I sent to Walker about "Intelligent Terminals"

A third, very important form of assistance is a little more subtle. Scanning a history list often stimulates recall by promoting associations. Long forgotten, implicit facts are frequently revived in one's mind by a chronological sequence of events of which one was the agent.

The archive file is a Hermes readable mailbox that contains every message ever referenced during an Inlat session, as well as copies of drafts of all messages ever composed by the user. Since the user will never operate on the archive mailbox himself (most likely will not even be aware of it--except for its massive size in his/her directory), nothing will ever be deleted from it, and hence each message in it can be given a unique message number that will never change (as opposed to the message numbers in normal mailboxes, which change all the time and cannot be used for identification between sessions). It is this unique number, henceforth called the 'archive number,' that is used to identify each message in the history list.

The history list is modeled somewhat after the Interlisp history list. When an operation is performed, relevant information about the operation is saved allowing the history list to be inspected, commands performed via a 'REDO' facility, and also, at least in principle, the ability to 'UNDO' something. Since the history list perpetuates indefinitely, there will be a certain problem with maintaining and searching it. It is expected that Interlisp-11 will have the ability to swap lists from files. In the meantime, the list is just a normal Lisp list.

The principal advantage of the history list, as of this writing, is the ability to look back to see what was done. The general format of a history list entry is indicated below. The list items are:

1] The command (or line) number

2] The command name (PRINT, FILE, SEND, etc.)

3] The date of the command

4] The command arguments (TEMPLATE, SEQUENCE, etc.)

5] List of archive numbers of messages referenced in the command, or subcommand tree

6] List of message numbers w.r.t. current mailbox

7] Input line exactly as typed by user

The subcommand tree in [5] denotes a secondary history list representing the subcommands involved in certain top-level commands such as COMPOSE, CREATE, EDIT, etc. The history list can be viewed as a chain of trees: each tree represents

a single top level command; the root of each tree is on the chain in sequential chronological order; the tree growing off of each root then represents all interactions and subtasks and subgoals used to accomplish the task initiated at the root ([2]). This corresponds to the subcommand structure within Hermes itself. It should be noted that there are several other 'history' lists maintained, one of which records Hermes (or Tutor) responses to initiatives on this primary history list.

Currently, a set of primitives exist for searching the history list. Before describing these primitives one other data table that is maintained should be mentioned. This is the 'message table.' It contains virtually all the information contained in a message except for the actual text of the message, i.e. all of the message header fields. This table is in one to one correspondence with the archive file. The archive numbers that reference the archive file also reference this table. As with the history list, this table is currently maintained in core, but for an operational system a swapping mechanism will be necessary due to its expected size.

As stated, the message table is an ordered list of messages, the order being exactly the same as that in the archive file. Each entry is a list, the first element of which is the MESSAGE-ID of the message*.(1) The remainder

(1) *The Message-Id field of a message is really the only way to 'name' a message, or refer to it uniquely within the

of the list has the exact same format as a Lisp property list, with the 'properties' being the header fields of the message, and the property 'values' being strings that are the respective values of the various fields in the message.

The MESSAGE-ID field of a message is the only way within Hermes of 'naming' a message. Accordingly, it is the key used for archiving and referencing all messages. Whenever a message is referenced its MESSAGE-ID is obtained (from Hermes) and that is saved as a single Lisp atom which is then hashed to find out whether or not it has been archived. If not, it is archived and an entry in the message table made. As indicated above, the message table entry contains virtually all the information about a message, except the actual text. Thus, to find out about a message, or back-reference on a message, only the message table need be consulted; to actually retrieve the text of the message, it may be necessary to unarchive it.

For example, to deal with the Inlat command

"WHERE DID I FILE THAT MESSAGE FROM AIGHES ABOUT QQSV?"

it would be necessary to search back through the history list to pull out the FILE commands; then using item [5] in the history list entries, searching until a message is found

---

normal mailbox. At present, we have determined empirically that between 1/3 and 1/2 of all incoming messages contain a Message-Id field. The Hermes people have indicated to us that at a future date they will automatically add Message-Id fields to all incoming messages that do not have this field.

whose property 'FROM:' is 'AIGHES.' Then item [4] of that history entry will contain the answer to the original question.

On the other hand, an Inlat command such as

"I WANT TO FORWARD THAT MESSAGE FROM BROWN ABOUT THE CAI CONFERENCE LAST MARCH TO SIGCAI"

would require the same sort of history lookup, but then would also require that the specified message be unarchived (assuming it has been deleted).

The primitives that currently exist have to do with isolating a portion of the history list to be searched (by specifying inclusive dates and/or commands of interest), and the obvious routines for pulling off specific fields in the commands. Since the message table entries are esssentially normal Lisp property lists, they can be searched in the obvious way.

APPENDIX

Below a sample session with the Inlat is reproduced. Inasmuch as the monitor has not yet been linked up to the NLF, everything shown was handled only by the monitor and Hermes. This is an actual typescript, slightly edited for readability. Everything in lower-case was printed by the system; the upper case characters were printed by either the system or the user. At the conclusion of the session there are some top level Lisp function calls that illustrate some of the 'history' primitives available.

In the Inlat typescript, each command line begins with a line number (that indexes the entries of the history list) followed by the Inlat prompt character which is the back-slash.

Typescript

34__INLAT(T]

Hermes/Inlat >>>> Preliminary version 0.0.2 (1-Jun-76 8:53PM)

Please be patient a moment while I start-up
Hermes over the Net and initialize ...

HERMES 2.6.5 1-Jul-76
BBN'rs! Check NEWS for vital information about CHECK-LOCAL!
Version 2.6.5, 7/13/76

   OK. Thanks for waiting.
89_\DAytime
Saturday, July 31, 1976 11:32:30-EDT

90_\Survey (messages) SInce TUEsday

56 27 Jul 1976 2141-EDT To: RUSTY 511
57 28 Jul 1976 1647-EDT AIGHES at BBN-TENEXE 1599
58 28 Jul 1976 2322-EDT FRENCH at BBN-TENEXA 1024
59 29 Jul 1976 1505-EDT BROWN at BBN-TENEXE 294

91_\Survey (messages) %-3-%_\FRom: BROWN + FRom: GRIGNETTI

38 13 May 1976 1135-EDT GRIGNETTI at BBN-TENEXA 604
49 12 Jul 1976 1046-EDT GRIGNETTI at BBN-TENEXE 1229
53 19 DEC 1975 1455-EST GRIGNETTI at BBN-TENEXA 1139
59 29 Jul 1976 1505-EDT BROWN at BBN-TENEXE 294

92_\Survey (messages) LAST FROM FRENCH

58 28 Jul 1976 2322-EDT FRENCH at BBN-TENEXA 1024
93_\STatus

59 messages in file <ASH>MSG.TXT;1 ; 1 unseen ; 0 deleted.

94_\Print (messages) 1-4 _\ FRom: GRIGNETTI (template) Ptemplate
     (to file) MARIO.MSG [New file]

95_\Survey (messages) LAST FROM BROWN

59 29 Jul 1976 1505-EDT BROWN at BBN-TENEXE 294

96_\DISplay (message) 59 (fields) SUbject:,CLAssification:,DAte:

```
Typescript


Subject:  A Sample message
Classification:   null field.
Date:  29 Jul 1976 1505-EDT
97_\Print (messages) 59 (template)  FULL-listing (to file)

Message 59;  294 chars
Mail from BBN-TENEXE rcvd at 29-Jul-76 1505-EDT
Date: 29 Jul 1976 1505-EDT
Sender: BROWN at BBN-TENEXE
Subject: A Sample message
From: BROWN at BBN-TENEXE
To: ASH
Message-ID: <[BBN-TENEXE]29-Jul-76 15:05:41.BROWN>
Hi

        This is just a sample message.
Bye ...
John
-------
_L

98_\DISplay (message) 59 (fields) SENder:

Sender:  BROWN at BBN-TENEXE

99_\Survey (messages) ON WEDnesday, FEB 17TH(=18-FEB-76)

100_\Survey (messages) ON TUES, FEB 16TH(=17-FEB-76)

17 17 FEB 1976 1003-EST RUBINSTEIN at BBN-TENEXD  824
18 17 FEB 1976 1003-EST RUBINSTEIN at BBN-TENEXD  886


101_\DISplay (message) 18 (fields) SUbject:,CLASI(=CLASSIFICATION:)

Subject:  Quote of the Week
Classification:   null field.


102_\Survey (messages) FROM MOOERS IN JANuary

 9 22 JAN 1976 0947-EST MOOERS at BBN-TENEXA  859
10 22 JAN 1976 1239-EST MOOERS at BBN-TENEXA  584

103_\DISplay (message) 10 (fields) SUBject:

Subject:  (((Describe command)))


104_\QUIT
 *** Resetting. Please wait for logout ...
```

Typescript

Stopping HERMES

Typescript

Below are some of the primitive functions shown as input to EVALQT. It is generally self-explanatory, but we will now give brief descriptions of the functions displayed.

HISTDATE[date(s)] returns a portion of the history list specified by the dates.

WHENN[linenumber] returns the date of a command.

HISTGET[linenumber] returns a single entry in the history list.

GETMSGS[arg] returns a list of archive numbers: if its argument is a line number, then it is the archive numbers of messages referenced by that command line; if its argument is a command, it searches through (a specified portion of) the history list and returns all messages that have been arguments to that command. E.g. line _#56, below, returns all messages that have been DELETEd.

ITEMS[arg] is like GETMSGS, but instead of returning the archive numbers, it returns a list of message table entries.

GTMSG[archivenumber] returns a single message table entry.

GETFIELD[archivenumber;field] returns a specific property of a message table entry.

GTMSGID[archivenumber] returns the Message-Id of an archived message.

There are various other primitves available, but these are the ones most easily illustrated by calls to EVALQT. Also, the values returned illustrate the physical lists maintained as history list entries and message table entries.

```
43_HISTDATE(TODAY]
((102 SURVEY 11269627725 (
*/FROM:% MOOERS/AFTER% 31-DEC-75/BEFORE% 1-FEB-76 TTY:) NIL NIL
("Survey FROM MOOERS IN JANUARY
")) (100 SURVEY 11269627425 (*/ON% 17-FEB-76 TTY:) NIL NIL (
"Survey ON TUES, FEB 16TH
")) (99 SURVEY 11269627395 (*/ON% 18-FEB-76 TTY:) NIL NIL (
"Survey ON WEDNESDAY , FEB 17TH
")) (100 SURVEY 11269627205 (*/AFTER% 30-JUN-31 TTY:) NIL NIL (
"Survey SINCE JUN 31
")) (99 SURVEY 11269627131 (*/BEFORE% 1-FEB-31/FROM:% GRIGNETTI
TTY:) NIL NIL ("Survey BEFORE FEBRUARY 31 FROM: GRIGNETTI
")) (97 PRINT 11269627009 (59 FULL-LISTING TTY:) (14) (59) (
"Print 59 FULL-LISTING
")) (95 SURVEY 11269626934 (59 TTY:) NIL NIL ("Survey LAST FROM BROWN
")) (94 PRINT 11269626882 (1,4,7,20 PTEMPLATE MARIO.MSG) (17 1 3
18) (20 7 4 1) ("Print 1-4  \ FROM:  GRIGNETTI  PTEMPLATE MARIO.MSG
")) (92 SURVEY 11269626617 (58 TTY:) NIL NIL ("Survey LAST FROM FRENCH
")) (91 SURVEY 11269626589 (38,49,53,59 TTY:) NIL NIL (
"Survey %%-3-%% \FROM:  BROWN + FROM:  GRIGNETTI
")) (90 SURVEY 11269626537 (*/AFTER% 27-JUL-76 TTY:) NIL NIL (
"Survey SINCE TUESDAY
")) (88 PRINT 11269615268 (59 FULL-LISTING TTY:) (14) (59) (
"Print 59 FULL-LISTING
")) (86 SURVEY 11269615135 (*/AFTER% 28-JUL-76 TTY:) NIL NIL (
"Survey SINCE WEDNESDAY
")))



44_WHENN(102]
"Saturday, July 31, 1976 11:52:45-EDT"

45_WHENN(86]
"Saturday, July 31, 1976 08:22:55-EDT"
```

Typescript

```
46__(PRINTDEF (HISTDATE "30-JUL-76") 5]
     ((84 SURVEY 11269374740 (49,53,59 TTY:)
       NIL NIL ("Survey %%-2-%%_\FROM BROWN + FROM GRIGNETTI
"))
      (83 SURVEY 11269374571 (*/AFTER% 28-JUL-76 TTY:)
       NIL NIL ("S SINCE WED
"))
      (81 SURVEY 11269374343 (*/AFTER% 29-FEB-76/BEFORE% 1-APR-76
         TTY:)
       NIL NIL ("Survey DURING MARCH
"))
      (80 SURVEY 11269374222 (24,35,38,49,53 TTY:)
       NIL NIL ("S %%-4-%% _\ FRIM GRIGNE
"))
      (78 PRINT 11269374017 (59 FULL-LISTING TTY:)
       (14)
       (59)
       ("Print LAST FROM BROWN FULL-
")))NIL



47__HISTGET(97]
(97 PRINT 11269627009 (59 FULL-LISTING TTY:) (14) (59) (
"Print 59 FULL-LISTING
"))



48__GETMSGS(97]
(14)


49__HISTGET(94]
(94 PRINT 11269626882 (1,4,7,20 PTEMPLATE MARIO.MSG) (17 1 3 18)
(20 7 4 1) ("Print 1-4 _\ FROM:  GRIGNETTI  PTEMPLATE MARIO.MSG
"))



50__GETMSGS(94]
(17 1 3 18)


51__WHENN(94]
"Saturday, July 31, 1976 11:38:42-EDT"
```

Typescript

```
52__ITEMS(97]
((<%[BBN-TENEXE%]29-Jul-76% 15:05:41.BROWN> COUNT 294 DATE:
"29 Jul 1976 1505-EDT" FROM: "BROWN at BBN-TENEXE" TO: "ASH" SUBJECT:
"A Sample message" SENDER: "BROWN at BBN-TENEXE"))


53__GETMSGS(97]
(14)


54__GTMSG(14]
(<%[BBN-TENEXE%]29-Jul-76% 15:05:41.BROWN> COUNT 294 DATE:
"29 Jul 1976 1505-EDT" FROM: "BROWN at BBN-TENEXE" TO: "ASH" SUBJECT:
"A Sample message" SENDER: "BROWN at BBN-TENEXE")


55__GTMSGID(14]
<%[BBN-TENEXE%]29-Jul-76% 15:05:41.BROWN>


56__GETMSGS(DELETE]
(7 2 4 5 6)


57__GETFIELD(7 SUBJECT:]
"GARBAGE"


59__GTMSG(7]
(<%[BBN-TENEXE%]14-Jul-76% 09:46:05.ASH> COUNT 51 DATE:
"14 Jul 1976 0946-EDT" FROM: "ASH at BBN-TENEXE" TO: "ASH " SUBJECT:
"GARBAGE" SENDER: "ASH at BBN-TENEXE")

60__DRIBBLE]
```

Previous Technical Reports

INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM

Technical Progress Report No. 1
18 Sept 1975 to 31 Mar 1976

Technical Progress Report No. 2
1 April 1976 to 30 June 1976